Switch Programing Guide

HP 3000 Computer Systems

HEWLETT PACKARD

HP Part No. 32650-90014
Printed in U.S.A. 19890901

U0989

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

There are many more manuals applicable to the HP 3000. A complete list may be found in every issue of the MPE V Communicator. Please contact your System Manager.

PREFACE

Although the manual provides basic as well as higher-level information, it is assumed that you have some familiarity with programming, with the HP 3000, or both. If you need further help or information, the following documentation will provide any in-depth discussions you may require:

- Using the HP 3000: *An Introduction to Interactive Programming (03000-90121)*
- *MPE File System Reference Manual (30000-90236)*
- *MPE V Intrinsics Reference Manual (32033-90007)*
- *MPE V Commands Reference Manual (32033-90006)*

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
`COMMAND`	Commands are shown in `CAPITAL LETTERS`. The names must contain no blanks and be delimited by a non-alphabetic character (usually a blank).
`KEYWORDS`	Literal keywords, which are entered optionally but exactly as specified, appear in `CAPITAL LETTERS`.
parameter	Required parameters, for which you must substitute a value, appear in ***bold italics***.
parameter	Optional parameters, for which you may substitute a value, appear in *standard italics*.
[ ]	An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: [A] [B] user may select A or B or neither. When brackets are nested, parameters in inner brackets can only be specified if parameters in outer brackets or comma place-holders are specified. Example: [*parm1*[,*parm2*[,*parm3*]]] may be entered as *parm1,parm2,parm3* or *parm1,,parm3* or *,,parm3* ,etc.

{ }	When several elements are stacked within braces the user *must* select one of these elements.
	
	Example: { A }
	{ B } user must select A or B or C.
	{ C }

...	An ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.
user input	In examples of interactive dialog, user input is underlined.
	Example: `NEW NAME?` ALPHA1
superscript^c	Control characters are indicated by a superscript^c.
	Example: `Y`^{c}. (Press Y and the CNTL key simultaneously.)
RETURN	RETURN indicates the carriage return key.

Contents

Figures

Tables

1

Introduction

Through a simple STORE/RESTORE process, you can move nonprivileged MPE V/E applications to MPE XL-based machines without modification, conversion, or recompilation.

Once applications are on an MPE XL-based system, you can choose how to execute them from the following four migration options involving two operating modes (Compatibility and Native):

- Emulation (Compatibility Mode)
- Object Code Translation (Compatibility Mode)
- Partial recompilation (Mixed Modes—Compatibility Mode and Native Mode)
- Full recompilation (Native Mode)

This manual focuses on the partial recompilation option. However, before beginning the discussion of mixed-mode operation, it is appropriate to give a brief overview of the two operating modes and the other three migration options.

Native Mode (NM)

Native Mode (NM), or direct execution on a 900 Series computer, allows use of the following HP Precision Architecture (HPPA) features:

- 32-bit words
- Contiguous memory
- Reduced instruction set
- 64-bit addressing

- Vastly increased stack/code/data space

The 900 Series systems run MPE XL, an extended version of the MPE V/E operating system.

Full Recompilation

After moving MPE V/E programs over to MPE XL, you will ultimately want to take advantage of the optimizing NM compilers. Recompilation can involve source code changes, especially if your application calls routines in languages not supported in Native Mode or uses MPE V/E features deleted from, or no longer available on, MPE XL. However, recompilation also affords the potential for performance gains since recompiled applications or routines execute directly in the NM environment.

Advantages

Programs running in Native Mode exploit the higher performance and expanded addressing capabilities of the HP Precision Architecture. Recompiled applications execute considerably faster than emulated or translated applications. The optimizing compilers written for MPE XL-based systems take high-level source code and compile it into NM object code. NM compiled programs run faster for the following reasons:

- They run directly on the 900 Series machine, rather than being emulated.
- They make fullest use of the more powerful HPPA instruction set.
- They take advantage of optimizations that the compilers produce by precisely scheduling and combining the HPPA instructions.

Considerations

For applications where source code is available, recompilation yields significant improvements in performance, simplicity, ease of maintenance, and supportability. When source code is not available or the effort of rewriting is not justified by the potential benefits, you can use the optimizing Object Code Translator (OCT) to translate MPE V/E-based programs and SL's. This translator produces significant performance gains over emulated programs run in Compatibility Mode, at the cost of code size expansion.

Compatibility Mode

Compatibility Mode (CM) is an emulation of the MPE V/E operating environment. Consequently, it is very compatible with HP 3000 object code from MPE V/E-based systems, allowing applications newly restored from MPE V/E-based systems to run immediately and transparently on 900 Series HP 3000s (with certain exceptions, as noted below in "Emulation and Translation Considerations").

Compatibility Mode is available to simplify the migration to the 900 Series of the HP 3000 family. You can move programs written in high-level languages (Pascal/V, FORTRAN 77/V, and COBOL II/V) to Native Mode by recompilation with an NM compiler. But you cannot recompile a program or procedure written in SPL in Native Mode, because HP does not provide an NM SPL compiler. For best performance, you should completely rewrite such code in a language having an NM compiler. However, if you must use an SPL program or procedure, it will run only in Compatibility Mode.

You have two options for running applications in the Compatibility Mode (CM) environment with few, if any, source code changes (discussed in more detail below):

- You can run applications directly in HP 3000 Compatibility Mode (using the Emulator).
- For better performance while still executing in Compatibility Mode, you can pass MPE V/E-based application program files through the Object Code Translator (OCT) and then run them.

Neither alternative requires any source code modification. A translated (processed by OCT) application program will generally execute somewhat faster than the same program executing under emulation. Recompiled programs should execute faster than either their translated or emulated versions.

Emulator

Under MPE XL, HP 3000 Compatibility Mode is implemented as an emulator for the CPU of MPE V/E-based systems and as an interface between supported MPE V/E intrinsic calls and MPE XL intrinsics. The Emulator functions as an object code interpreter. 900 Series machines have several features that make them good interpreters:

- Simplicity of the instruction set
- Simplicity of control paths

The HP 3000 object code Emulator works by interpreting (reading and initiating appropriate machine activity), at run time, each MPE V/E-based instruction into a functionally equivalent sequence of 900 Series instructions. Because the machine instruction set on MPE XL-based systems has been simplified from that found on MPE V/E-based systems, it often takes several 900 Series instructions to emulate a single MPE V/E-based instruction.

Thus, even though interpreted execution is slower than direct execution of the same instructions, the advanced features of the 900 Series machine yield performance under emulation (running in Compatibility Mode) similar to that of a HP 3000 Series 70.

Object Code Translator

The Object Code Translator (OCT) converts the MPE V/E-based instructions in a CM program file into 900 Series instructions. This Object Code Translator is valuable for programs whose source is not available and as a low-cost speed-up of compute-bound CM programs.

You invoke the Object Code Translator by means of the `:OCTCOMP` command. OCT not only translates the CM instructions to NM instructions; it also does whatever low-level optimizations are possible without reference to the program's source. OCT performs an optimized translation of code once and saves the result, eliminating the need for reading and decoding each time the program is run. OCT produces an output file, either permanent or temporary depending on the option selected. Consequently, overhead during execution is greatly reduced, but the translated MPE V/E program or SL will require greater disc space (currently up to 10 times more space; the actual file size usually expands by about 3-4 times) than the original program file.

Emulation and Translation Considerations

A code size expansion occurs when translating MPE V/E-based object code to its 900 Series equivalent using the Object Code Translator. You have a choice between higher performance execution using the Object Code Translator or more efficient disc space utilization using the Emulator.

Another tradeoff involves the ease of debugging. Since original MPE V/E program files are unchanged when emulated, they can be debugged using the CM portion of the MPE XL Debugger. In contrast, MPE V/E programs that have been translated by means of the Object Code Translator consist of optimized 900 Series instructions. This requires use of the NM debugger on mechanically-translated object code and is more difficult than using the NM debugger on CM programs.

Both the Emulator and the Object Code Translator need to know MPE V/E-based instructions and stack architecture. The Emulator creates MPE V/E-based program and stack structures when a CM program is run. Object code compatibility is provided by exactly emulating the MPE V/E-based environment. You are subject to the same addressing and stack limitations found on MPE V/E-based systems. OCT executes in Native Mode on a CM program file. Its output, a translated program, executes in Compatibility Mode on CM data, thus still having the addressing and stack limitations. If you require additional features and performance, you need to migrate your application(s) to Native Mode.

You can use Compatibility Mode not only to run MPE V/E applications immediately following migration, but also, over the long term, to execute those applications that perform acceptably on MPE V/E-based systems today.

Note Applications running in Compatibility Mode must not execute privileged instructions and must call only documented, callable MPE V/E or subsystem intrinsics. However, you can enter Privileged Mode and call MPE V/E privileged intrinsics from Compatibility Mode.

Caution The normal checks and limitations that apply to standard users in MPE XL are bypassed in Privileged Mode. A Privileged Mode program could destroy file integrity, including the MPE

XL operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of Privileged Mode code. This service, which is not provided under the standard service contract, is available on a time and materials billing basis. Hewlett-Packard will not support, correct, or attend to any modification of the MPE XL operating system software.

Summarizing NM-only and CM-only Execution

Figure 1-1 illustrates the place of emulation, translation, and full recompilation within the Native Mode (NM) operating environment. The figure shows three processes running independently under MPE XL:

- Process 1 is a CM program (obtained from a STORE/RESTORE) running under emulation. Emulation is a feature of the MPE XL operating system that uses existing program files and data to produce the same results as if the program ran on an MPE V/E-based system.
- Process 2 is a CM program that has been translated to achieve a performance boost. This translation involves converting portions of a CM program file to NM object code and appending that code to the file. The performance gain is achieved by escaping from the Emulator to this NM code during execution of the CM program. Translated programs are still subject to all CM limitations and still execute in Compatibility Mode.
- Process 3 is an NM program (produced by full recompilation). This program can take advantage of the full MPE XL feature set, including the optimizing NM compilers and the performance improvement resulting from direct execution of the HP Precision Architecture instruction set.

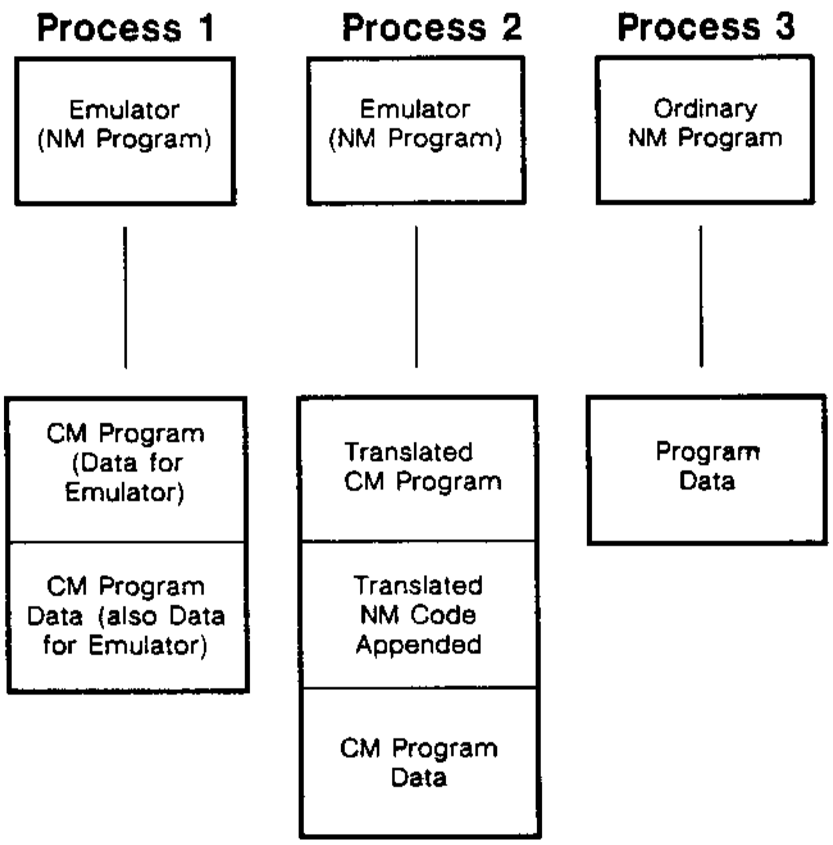


Figure 1-1. MPE XL Execution Environment

The translated program in Figure 1-1 is the result of a one-time translation process illustrated in Figure 1-2. The translation process is invoked by issuing the command: `:OCTCOMP` *CMprogname*.

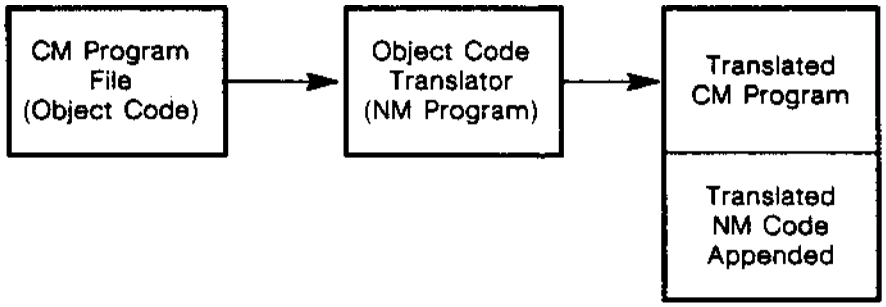

CM Program File (Object Code)
Object Code Translator (NM Program)
Translated CM Program
Translated NM Code Appended

Figure 1-2. OCT Translation Process

Mixed-mode Procedure Calling (Partial Recompilation)

Partial recompilation, possibly requiring rewriting, employs the Switch subsystem (Switch) to make mixed-mode procedure calls. The MPE XL Switch subsystem (Switch) provides the ability for programs executing in one mode to call Compatibility Mode (CM) Segmented Library (SL) or Native Mode (NM) Executable Library (XL) procedures that reside in the opposite mode. For instance, users with high-level NM applications that call low-level CM SL procedures (COBOL calling SPL Segmented Library code, for example) can recompile the high-level portion using the appropriate compiler and, using Switch, can call CM SL procedures. With this capability, you get the advantage of Native Mode (large code and data space and performance improvements) in the recompiled portion of your application, without having to rewrite SL procedures in a supported NM language.

Figure 1-3 illustrates the place of Switch in the NM environment.

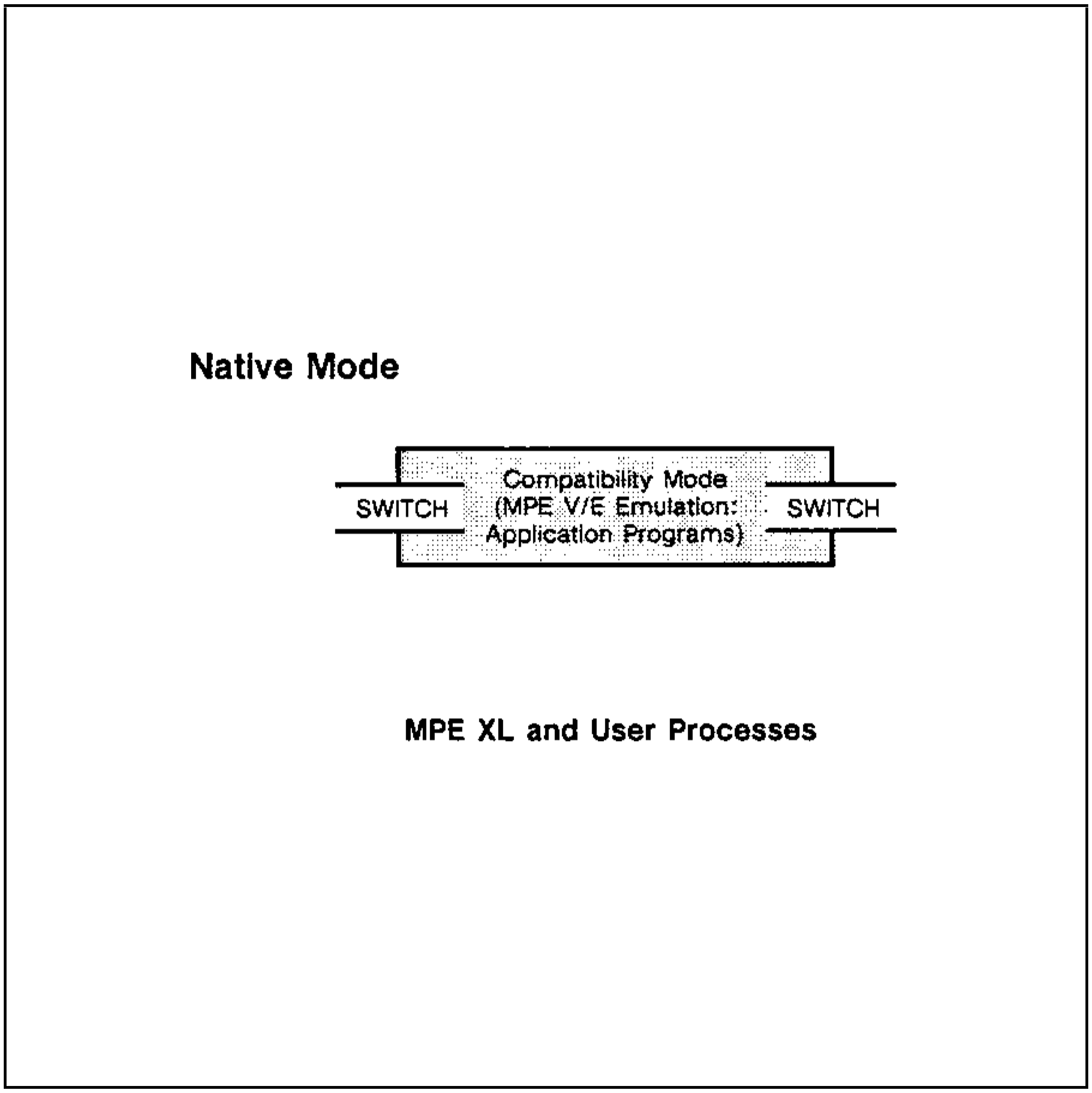


Figure 1-3. Switch in MPE XL Execution Environment

Mixed-mode Overview

MPE XL-based systems provide two operating modes (Compatibility Mode and Native Mode) and a variety of execution options:

1. You can run CM programs (directly after being restored from your MPE V/E STORE tape), and, from that code, call procedures that are located in a CM SL. In this instance, all execution is within Compatibility Mode, and no apparent mode switching on the part of your program is involved. Figure 1-4 illustrates this model.

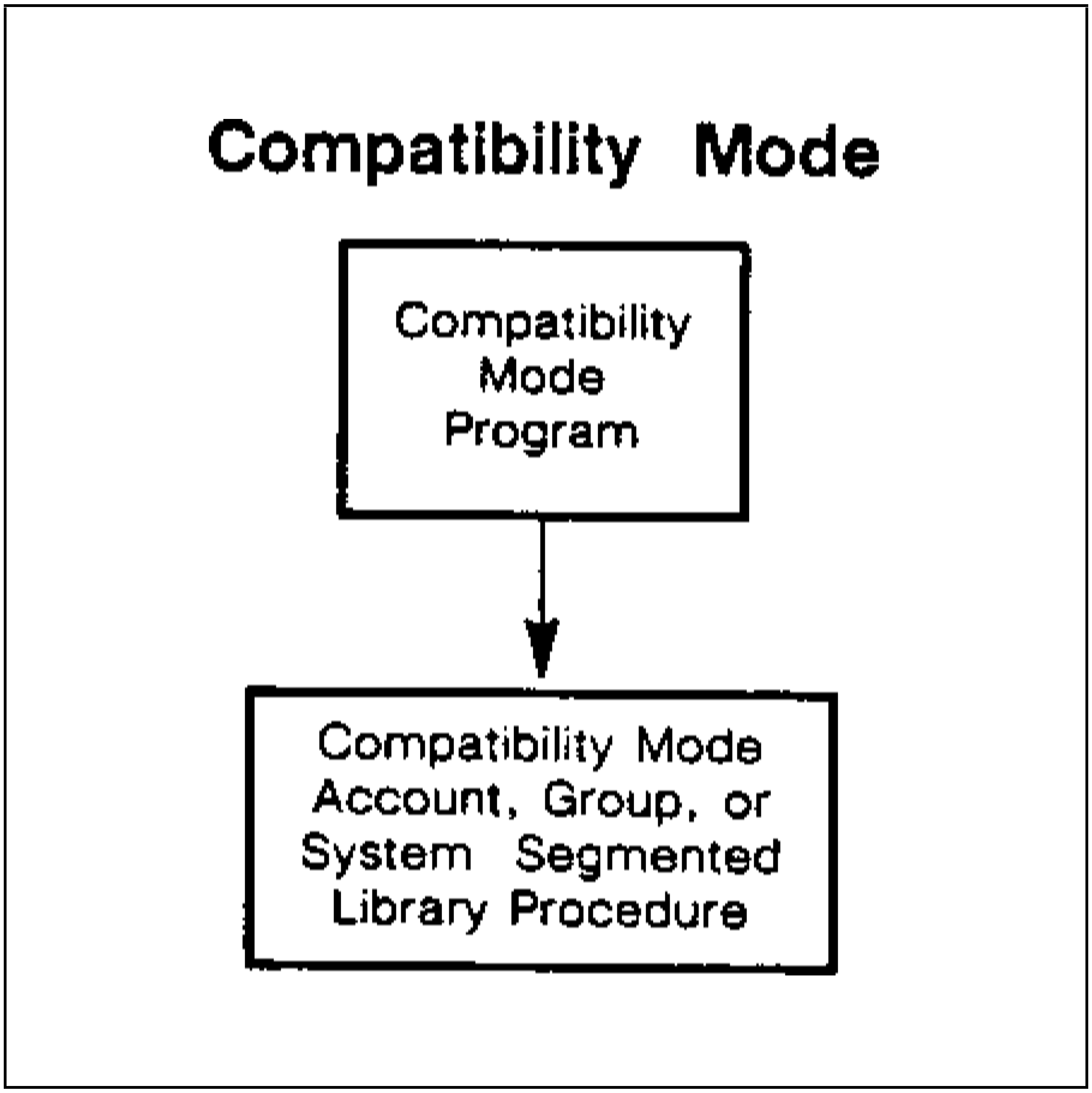

Compatibility Mode
Compatibility Mode Program
Compatibility Mode Account, Group, or System Segmented Library Procedure

Figure 1-4. Compatibility Mode Execution Model

2. You can run NM programs (after full recompilation), and, from that code, call procedures that are located in an NM Executable Library. In this instance, all execution is within Native Mode, and again no apparent mode switching on the part of your program is involved. This is demonstrated in Figure 1-5.

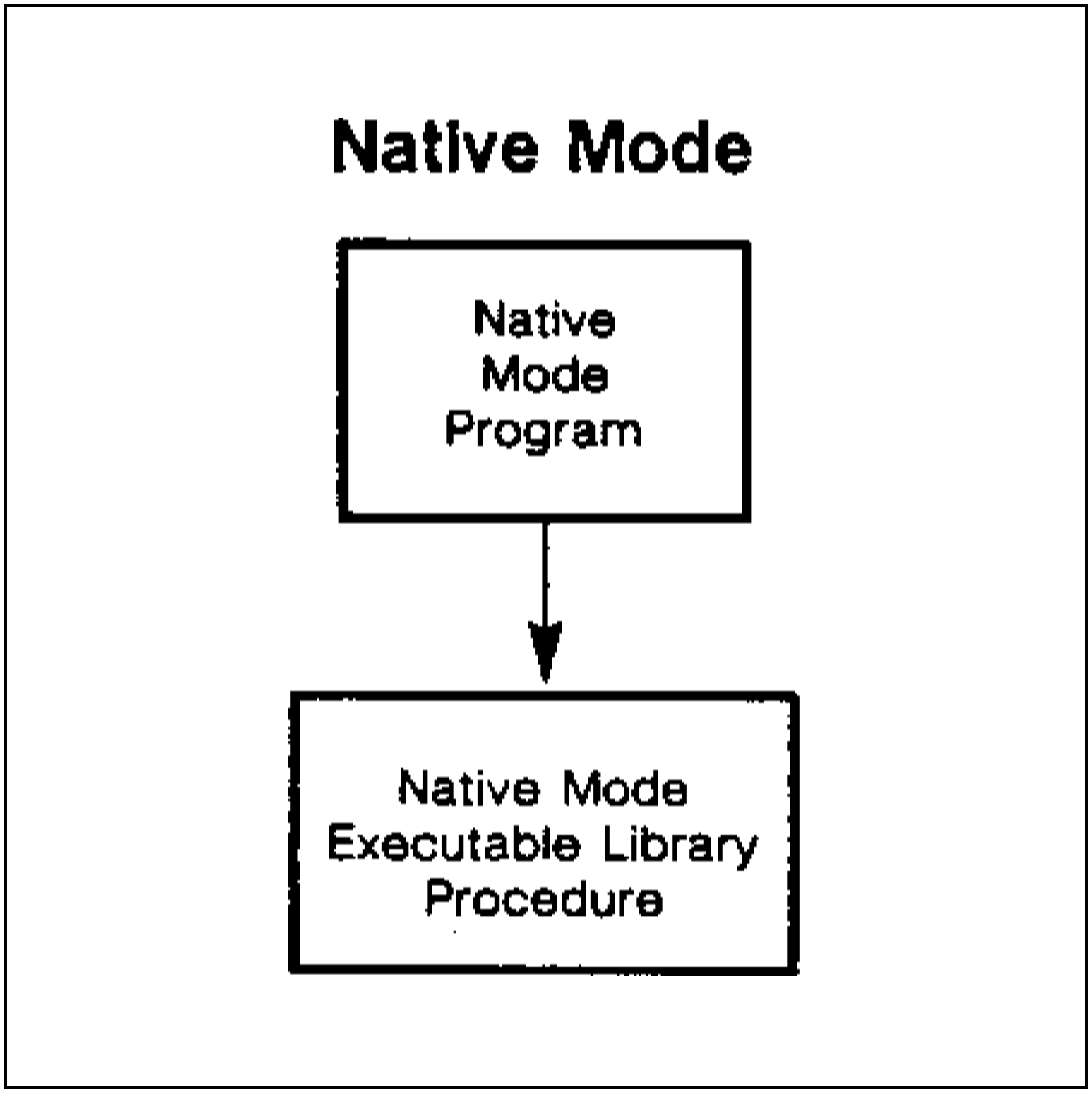


Figure 1-5. Native Mode Execution Model

3. When running NM code you can call procedures located in a CM SL, or when running CM code you can call procedures located in an NM Executable Library. These situations involve mixed-mode procedure calls, and Switch provides such capability. Figures 1-6 and 1-7 symbolize this mixed-mode calling feature.

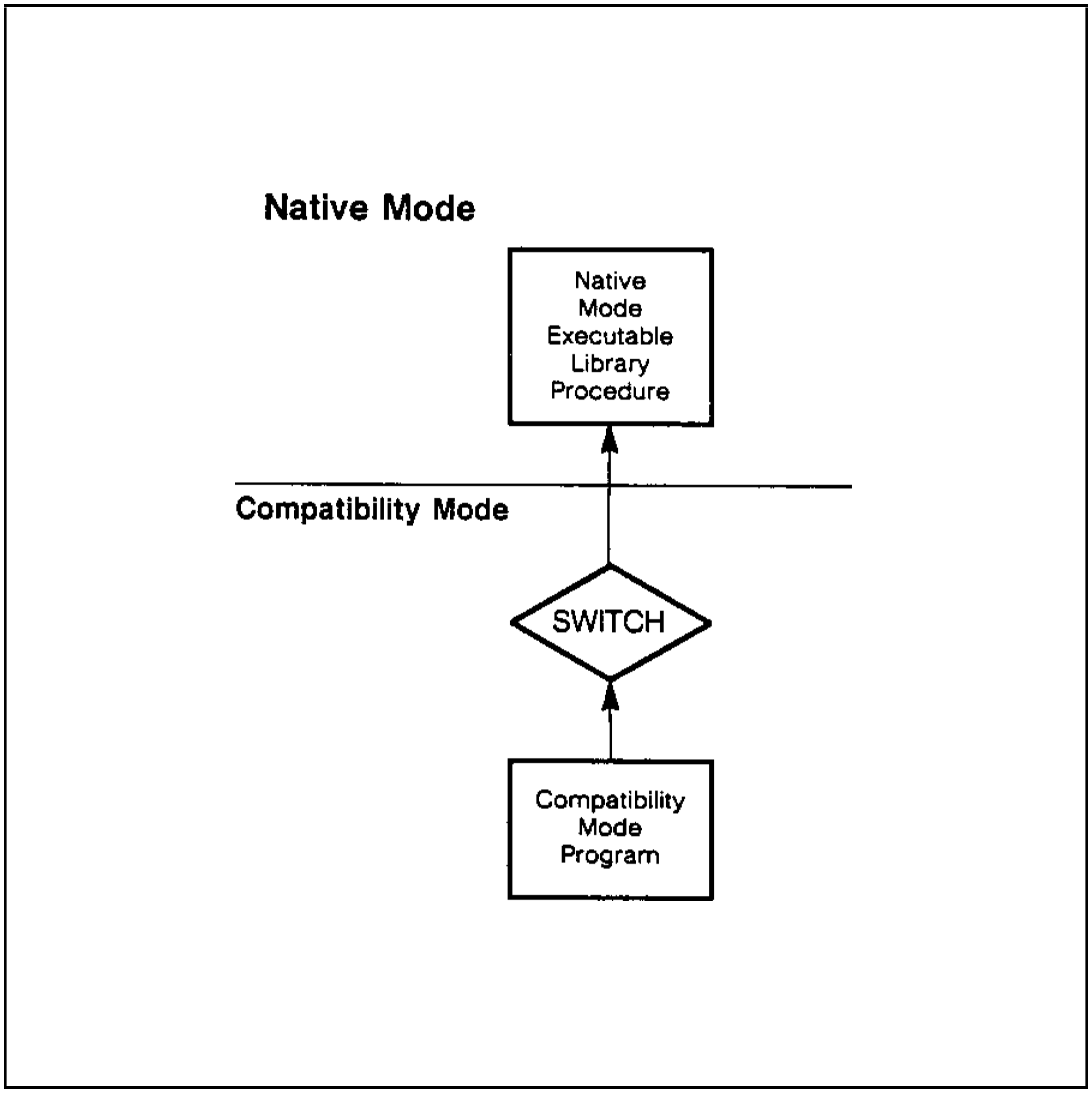

Native Mode
Native Mode Executable Library Procedure
Compatibility Mode
SWITCH
Compatibility Mode Program

Figure 1-6. CM—> NM Mixed-mode Execution Model

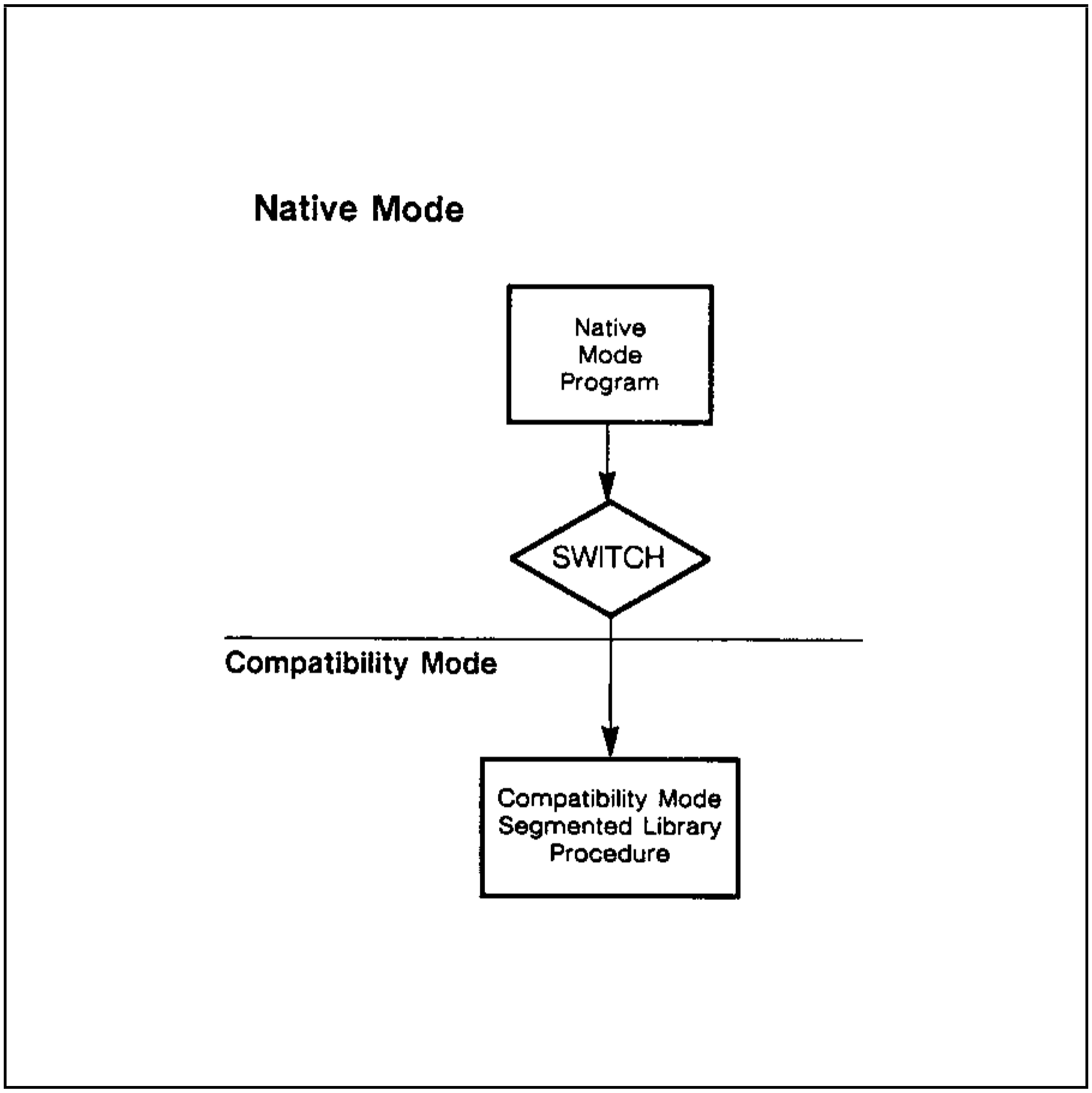


Figure 1-7. NM—> CM Mixed-mode Execution Model

Note

Combinations of mixed-mode procedure calls are also possible. For example, your NM program could use Switch to call a CM SL procedure that, in turn, could call an NM Executable Library procedure via Switch (NM—> CM—> NM), and so on. Conversely, your CM program could use Switch to call an NM Executable Library procedure that, in turn, could call a CM SL procedure via Switch (CM—> NM—> CM), and so on. However, you should carefully consider use of this feature since you pay an overhead penalty on every switch.

Mixed-mode Situations

If you have existing applications written in a high-level language that call user-written CM SL procedures that are written in SPL or other languages not supported by Hewlett-Packard in Native Mode, and you want to recompile these applications in Native Mode, then you will need to use Switch. To take full advantage of the features and performance of 900 Series machines, it is necessary to recompile existing applications entirely in Native Mode. However, a number of factors may make full recompilation infeasible. Among these are the following:

- Size of the application
- Availability of shared procedures
- Language of implementation (SPL, for example)
- Desired performance
- Resources available for migration
- Availability of source code of CM SL procedures

Such factors can lead you to split the application into two different modes of execution, Compatibility Mode and Native Mode, and migrate by phases, until you have migrated the entire application into Native Mode.

Or, mixed-mode execution may be your target state. You may need to make mixed-mode procedure calls in the NM—> CM direction if your high-level applications running on MPE V/E-based systems make calls to SPL procedures in order to take advantage of the special features accessible from that language. Because SPL is close to the HP 3000 architecture of earlier series, it is able to take particular advantage of that architecture. However, SPL is so architecture-dependent that it has not been migrated to Native Mode. Consequently, the user of SPL procedures is faced with two options:

1. Rewrite all SPL procedures in higher-level languages that can be compiled directly in Native Mode.
2. If rewriting is not feasible or the procedure is not performance-sensitive, use Switch to continue to call such procedures but call them from code that has been migrated to (recompiled in) Native Mode.

Other factors may motivate CM—> NM switching. A common situation involves the most performance-sensitive portions of your application. These are probably the first pieces of code you want to recompile or rewrite. Mixed-mode procedure calling allows you to migrate those portions to Native Mode to obtain the performance gains, while still running the remainder of the application in Compatibility Mode. This provides a continual, incremental upgrade path. Be sure that the overhead of the Switch is outweighed by the performance advantage of running that code in Native Mode.

Relationship to Migration

When you move your application or subsystem from MPE V/E- to MPE XL-based systems, you can carry out this migration in a phased manner. Phased migration refers to the ability to convert your application in steps from 100% CM execution to, potentially, 100% NM execution.

Because Hewlett-Packard has provided object code compatibility, most applications can be restored and execute in Compatibility Mode without changes. You can then migrate a portion of your application to Native Mode to take advantage of increased performance and added features. You can choose to migrate your database, application programs, or application libraries independently of one another. It is this capability to migrate portions of an application over time that makes up the phased migration feature of MPE XL.

Since the HP Precision Architecture is different from that of non-MPE XL-based HP 3000 family members, users of the 900 Series face certain differences when migrating to the MPE XL environment. Furthermore, migration does not end when an MPE XL-based system is installed. In many instances, Series 900 systems will be used in both networks and environments alongside MPE V/E-based systems. The transfer of information between these types of systems and the attendant need to convert between MPE V/E- and MPE XL-based data may make migration an ongoing concern.

Cost/Performance Factors

As noted previously, you have the ability to run applications in Compatibility Mode, Native Mode, or mixed modes. To some extent, the choice of how to mix execution modes may involve the question of how to achieve the best possible performance. The following recommendations are intended to guide you in making a decision appropriate to your circumstances:

- Find the procedure that uses the most resources.
- If that procedure is CPU-intensive (for example, doing calculations or character comparisons), you may benefit from converting it to Native Mode and accessing it via Switch.
- If that procedure is a special-purpose SPL technique to make your application bigger or faster, look for an NM feature (such as the large data space or mapped I/O) that can take the place of the SPL procedure.
- Take measurements to be sure that the Switch overhead does not outweigh the performance improvement you gain from converting to an NM procedure.

Switch Subsystem

The Switch subsystem is an integral part of the MPE XL operating system. Switch consists of a set of intrinsics that provide the following services:

- Ability to call subsystem, intrinsic, or user-written CM SL procedures from NM system or user code
- Ability to call subsystem, intrinsic, or user-written NM XL procedures from CM system or user code

Note MPE V/E intrinsics (except for those no longer supported) are directly callable from both Compatibility Mode and Native Mode. You do not need to code any Switch calls to access these intrinsics.

Figures 1-8 and 1-9 demonstrate the role of the Switch intrinsics in the process of carrying out mixed-mode procedure calls:

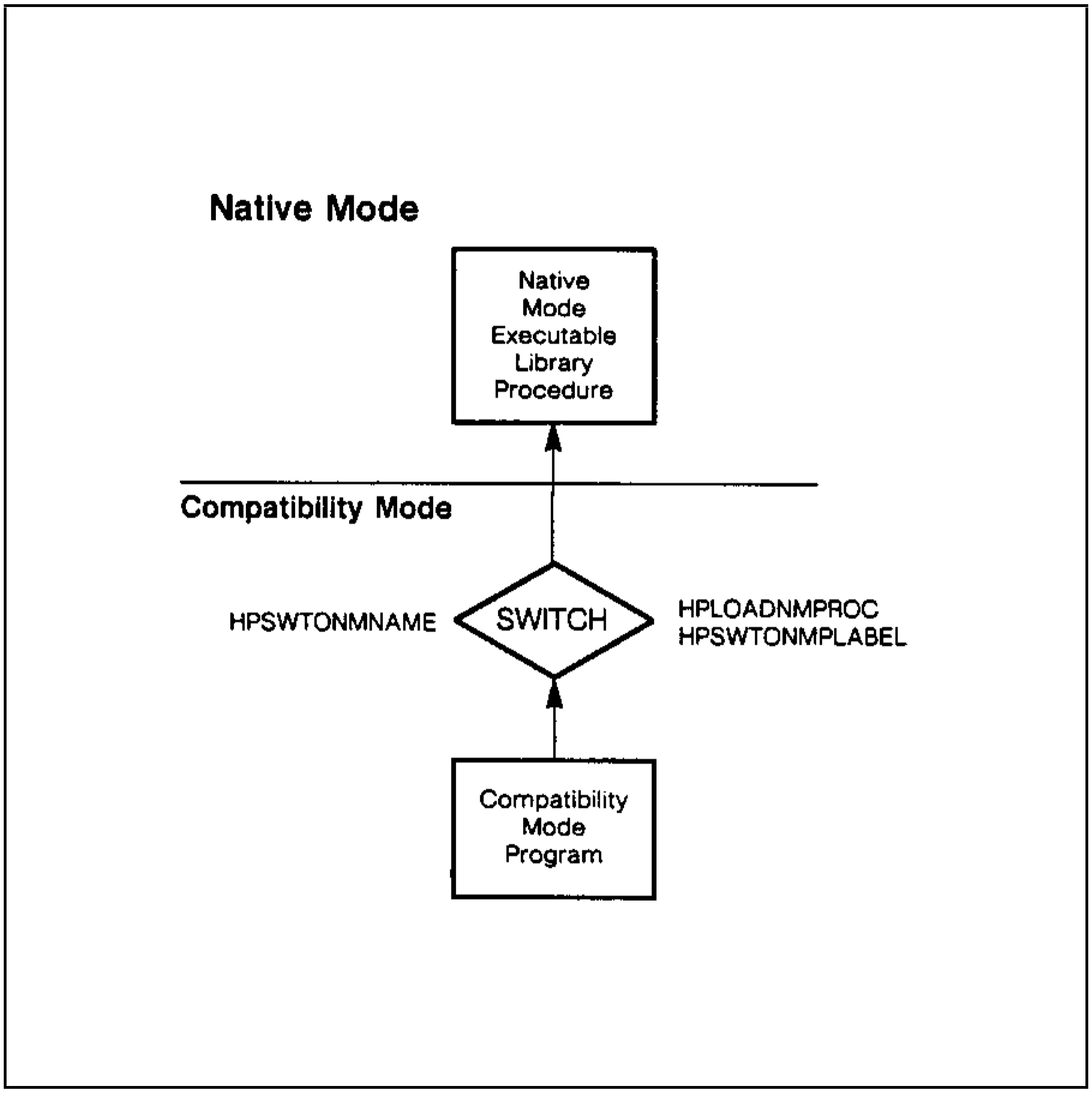


Figure 1-8. Role of Switch Intrinsics in CM—> NM Switch

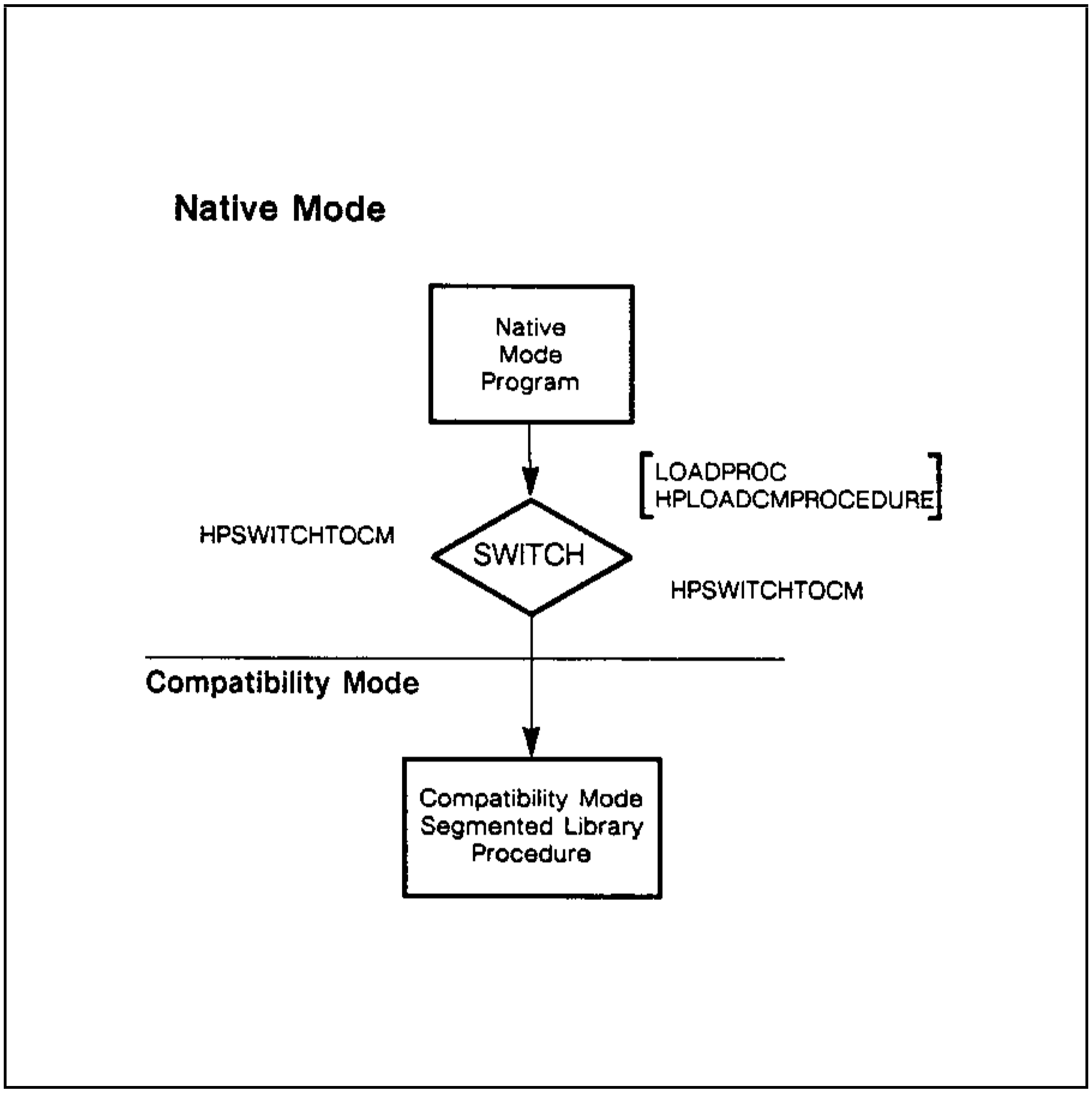


Figure 1-9. Role of Switch Intrinsics in NM—> CM Switch

Purpose

Due to architectural differences between MPE V/E- and MPE XL-based systems, programs that switch between CM and NM execution may encounter differences in data representation. The primary purpose of Switch is to resolve this situation by providing mixed-mode execution access and parameter translation between Compatibility Mode and Native Mode.

Invoking Switch

One way to invoke Switch is to use an intermediate, user-written procedure that sets up variables and calls Switch to translate these data references whenever the program changes execution mode. This intermediate procedure is called a Switch stub. The primary purpose of a stub is to provide transparency to the calling procedure/program and thereby avoid source changes in the original code. To do this, the stub must have the same name as the original procedure.

You can use the SWitch Assist Tool (SWAT) to automatically generate NM-to-CM stub procedures, or you can write your own Switch stubs. Refer to Chapter 2 for information on SWAT and to Chapters 3-5 for information on the Switch subsystem and the process of writing your own stubs.

Another way to invoke Switch is to code the Switch call, and its attendant setting up of variables and error checking, directly in line in the calling procedure. Although this involves modification of your program, there are factors that may lead you to consider this alternative. For example, every time you call a procedure or function, the call and the entry and exit code add to execution overhead. This overhead could increase execution time if it occurred in a critical section of your program.

In-line switch code may be appropriate for other-mode procedures that should execute quickly and/or are executed often (for example, in a loop). If an other-mode procedure is referenced in many places, then the convenience of the stub can justify the added overhead of a procedure call. Another factor influencing the choice between stubs and in-line switches is whether data setups can be reused. If not, the in-line switch is not justified. Finally, if backward source compatibility is important to you, you should use Switch stubs. To compile an application or module on an MPE V/E system, removing the stub from the compile procedures is all that is required.

Switch to CM Overview

Figure 1-10 represents the process of mixed-mode procedure calling in the NM—> CM direction.

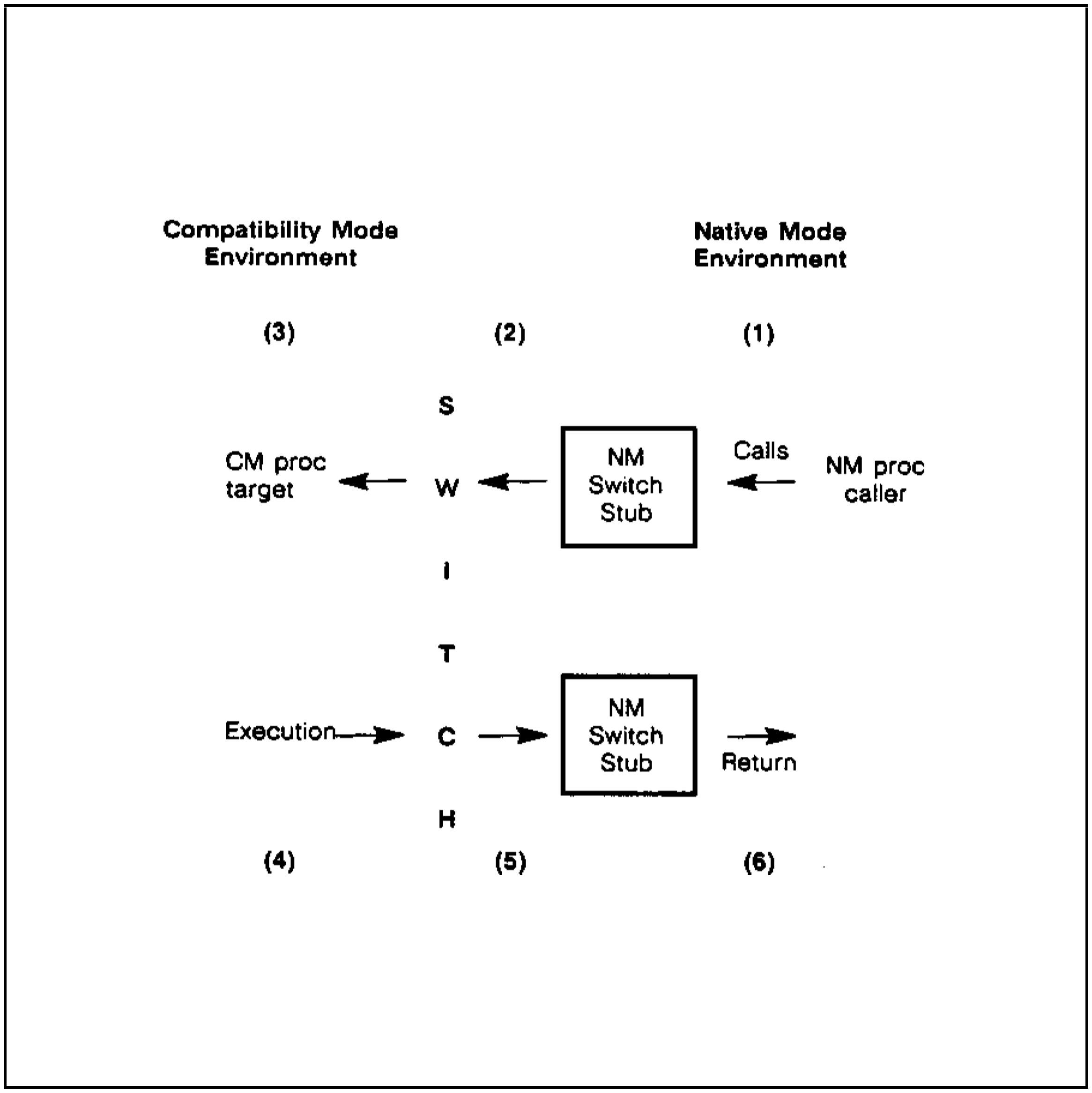


Figure 1-10. Mixed-mode Procedure Call: NM—> CM

A mixed-mode procedure call in the direction NM—> CM involves the following steps, as indicated in Figure 1-10:

1. The NM code needing to access a CM routine calls a Switch intrinsic, either by means of in-line code or a Switch stub.
2. The in-line code or NM Switch stub sets up the data structures and parameters required by Switch and makes a call to the `HPSWITCHTOCM` intrinsic (call by name), or the `HPLOADCMPROCEDURE` and `HPSWITCHTOCM` intrinsics (call by plabel).
3. `HPSWITCHTOCM` calls the CM routine and passes the parameters, translating addresses and/or copying data to Compatibility Mode as required by their type, size, and location.
4. The CM routine executes using the passed parameters and returns its data values to `HPSWITCHTOCM`.
5. `HPSWITCHTOCM` receives the data values returned by the CM routine, back-translating addresses and recopying data as needed.
6. `HPSWITCHTOCM` returns the translated values to the caller. The stub or in-line code checks whether the Switch operation was successful and then control returns to the NM routine.

Note `HPSWITCHTOCM` is the system intrinsic that makes NM parameters addressable to Compatibility Mode, changes the execution mode, and invokes the CM routine. `HPSWITCHTOCM` and `HPLOADCMPROCEDURE` reside in the NM system library, `NL.PUB.SYS`.

Switch to NM Overview

Figure 1-11 represents the process of mixed-mode procedure calling in the CM—> NM direction.

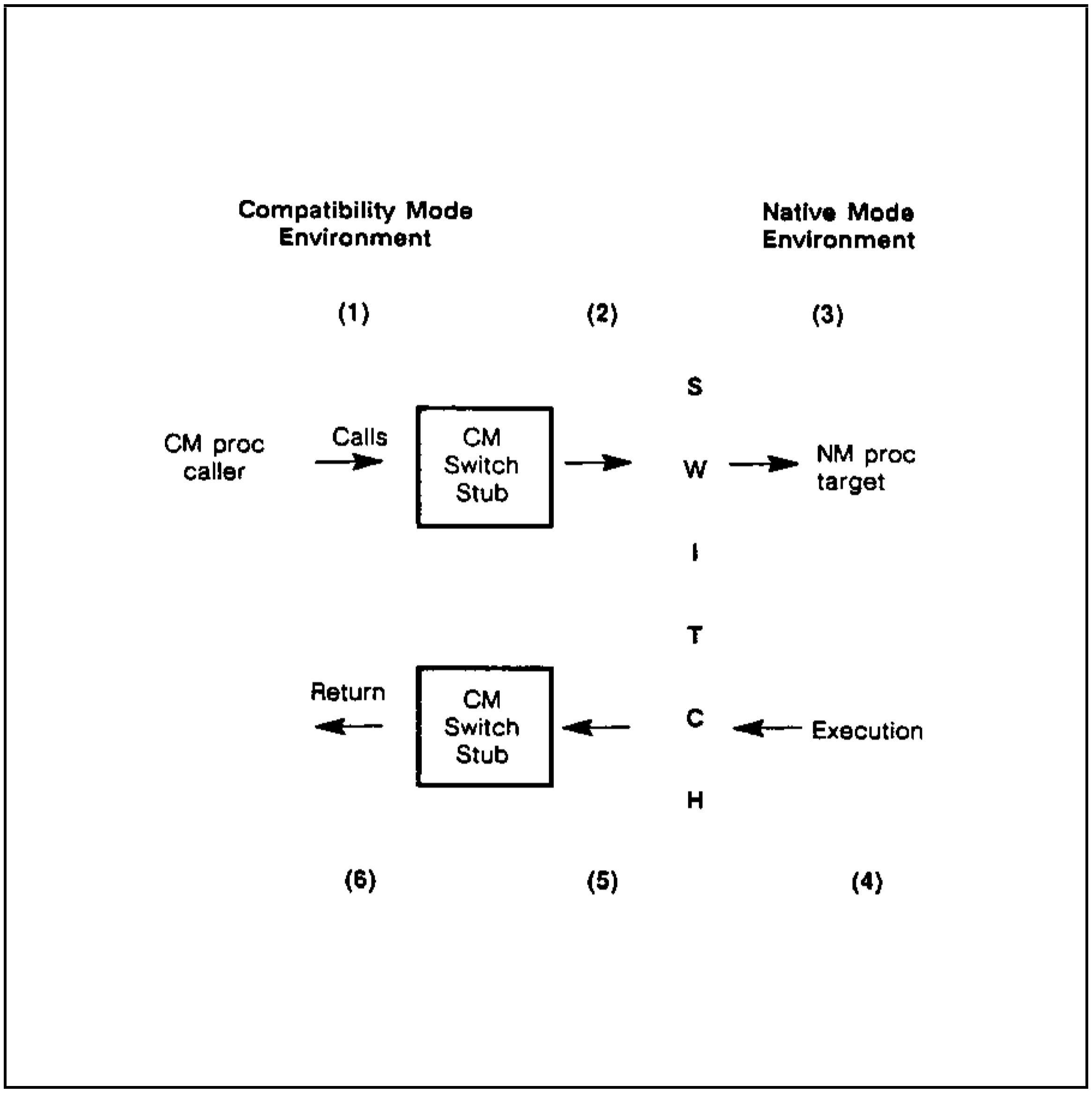


Figure 1-11. Mixed-mode Procedure Call: CM—> NM

A mixed-mode procedure call in the direction CM—> NM involves the following steps, as indicated in Figure 1-11:

1. The CM code needing to access an NM routine calls Switch, either by means of in-line code or a Switch stub.
2. The in-line code or CM Switch stub sets up the data structures and parameters required by Switch and makes a call to either the `HPSWTONMNAME` intrinsic or the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics.
3. `HPSWTONMNAME` or `HPSWTONMPLABEL` calls the NM routine, after preparing the NM registers as though the call was being made from Native Mode.
4. The NM routine executes as though called from Native Mode and returns the functional return value (if any) to the calling Switch intrinsic. Reference parameters are modified by the NM routine.
5. If the NM routine has a functional return value, `HPSWTONMNAME` or `HPSWTONMPLABEL` copies the value back to the CM stack.
6. The stub or in-line code checks whether the Switch operation was successful and then control returns to the CM routine.

Note `HPSWTONMNAME` or a combination of `HPLOADNMPROC` and `HPSWTONMPLABEL` are the system intrinsics called to pass CM values and parameters to Native Mode, to change the execution mode, and to invoke an NM routine. `HPSWTONMNAME`, `HPLOADNMPROC`, and `HPSWTONMPLABEL` reside in the CM system library, `SL.PUB.SYS`. The only difference between using `HPSWTONMNAME` and `HPLOADNMPROC/HPSWTONMPLABEL` is the manner in which the target routine is identified. For details on when and why to use names versus plabels, see Chapter 4.

Switch Stubs

A Switch stub is a routine that acts as a facade, hiding the complexity of the actual Switch call. The primary purpose of Switch stubs is to provide applications that must call routines residing in an other-mode library with transparent access to Switch.

Transparency for the application making the mixed-mode call means that the recompiled calling program usually does not require source changes to call procedures from its original mode. The Switch stub makes this transparency possible due to the following factors:

- The Switch stub procedure name is identical to the original procedure name.
- The stub parameters and their types are equivalent to those of the original procedure.
- The Switch stub sets up the special parameters required by the Switch intrinsic (`HPSWITCHTOCM`, `HPSWTONMNAME`, or `HPLOADNMPROC` and `HPSWTONMPLABEL`).
- The stub makes the call to the intrinsic.
- The Switch stub handles the necessary return parameters (if any).

In this way, an application program that must now call an other-mode routine can call an identically named procedure in its own mode and thereby avoid making changes in the application source code. The only change required is the addition of a stub procedure to a CM Segmented Library (SL) or an NM Executable Library (XL).

Note Sometimes the calling procedure must change to allow it to do sophisticated recovery from error situations. Data alignment is an ongoing concern and possible cause of change.

Operation

The Switch stub is responsible for doing the following:

- Setting up the data structures and parameters required by Switch intrinsics
- Making the call to the Switch intrinsic, which translates and copies parameters as required and triggers the change in operating mode
- Retranslating the parameters that the mixed-mode procedure call returns through Switch
- Setting the condition code on return from the NM routine (CM—> NM only)
- Error handling

Switch makes the data and parameters of the calling procedure understandable to the called procedure, and vice versa. Use of a stub concentrates the programming effort to call the Switch intrinsics in one place. All calls to the other-mode routine are routed through the stub, so there is no need for direct calls to the Switch intrinsics by the application itself.

Use

For the mixed-mode call to succeed, three pieces of code are needed:

- Calling routine
- Switch stub in same mode (NM or CM) as the caller
- Other-mode routine in library of mode opposite the caller (NM Executable Library or CM Segmented Library)

When a program is loaded (run), the loader cannot bind any unresolved external references to procedures residing in an other-mode library. Whenever an external reference is encountered, the loader searches the specified libraries of the calling mode to resolve that reference. If the external reference cannot be resolved by a search of those current mode libraries, then the load fails.

For example, an NM program that refers to an external procedure residing in a Compatibility Mode (CM) Segmented Library (SL) cannot be loaded. The CM SL's are not searched by the NM Loader because all entry points in a program file, executable library, or object library must be in the same mode. Neither loader resolves an external reference made in one mode to an executable library in the opposite mode, because object file formats for the two processors are very different.

Explicit calls to Switch intrinsics resolve such external references by identifying the file from which to load the procedure and providing the information needed to translate parameters. This is possible because one of the parameters passed to the Switch intrinsic designates the library in which the external procedure can be found. The Switch stub, among other things, calls the appropriate Switch intrinsic (`HPSWITCHTOCM`, `HPSWTONMNAME`, or `HPSWTONMPLABEL`), which, in turn, calls the other-mode procedure.

After you produce and compile a Switch stub, use the Linker or Segmenter (depending on the mode of the caller) to place the object code in the appropriate system library (NM Executable Library or CM Segmented Library) or to incorporate the stub code into the calling program.

Refer to the *MPE XL Commands Reference Manual* (32650-90003) for information on the commands used to invoke the various NM and CM compilers.

Refer to the *MPE Segmenter Reference Manual* (30000-90011) for information on CM Segmented Libraries.

Refer to the *Link Editor/XL Reference Manual* (32650-90030) for information on linking object code modules into NM program files and installing object modules in NM Executable Libraries.

Impact on Users

Switch is used in the following areas:

- Phased migration of existing applications
- Recompilation into Native Mode of existing source code that contains calls to user-written CM SL procedures
- New NM applications that call user-written CM SL procedures

Members of the following groups will make use of Switch:

1. Suppliers of application programs and subsystems who want to migrate that software to Native Mode in a phased approach
2. Users who choose to recompile existing source code with MPE XL-based NM compilers (HP Pascal/XL, HP FORTRAN 77/XL, COBOL II/XL) and whose original source contains calls to user-written CM procedures that remain in CM SLs and where rewriting these procedures in an NM language is impractical or impossible
3. Developers of new NM programs that require the services of non-intrinsic CM procedures
4. Users with performance-sensitive MPE V/E SL procedures who want to recompile or rewrite those procedures in Native Mode to get the best performance, but also want CM programs to continue to call the procedures
5. Suppliers of application software who need to provide access to their routines from both modes

If you fall into the first category, you should do the following:

- Generate NM—> CM Switch calls (coded in-line or in Switch stubs) to non-intrinsic CM procedures
- Generate CM—> NM Switch calls (coded in-line or in Switch stubs) from CM procedures to NM procedures
- Incorporate into an MPE XL Executable Library any new NM procedures that CM procedures call via Switch
- Understand the performance factors that affect mixed-mode procedure calls

Members of the second category should do the following:

- Recompile application programs written in Pascal, FORTRAN, or COBOL with the appropriate NM compiler
- Generate Switch calls (either by means of in-line code or Switch stubs linked with the converted programs) to user-written CM procedures that remain in CM SLs

Those in the third category should do the following:

- Generate Switch calls (either via in-line code or Switch stubs linked with the new NM programs) to CM procedures

Those in the fourth category should do the following:

- Write NM procedures to replace performance-critical CM procedures and place them in an Executable Library
- Generate CM—> NM Switch calls (through in-line code or Switch stubs) to these NM procedures

Those in the final category should do the following:

- Generate the appropriate NM—> CM Switch calls (coded in-line or in Switch stubs)
- Generate the appropriate CM—> NM Switch calls (coded in-line or in Switch stubs)
- Incorporate into an MPE XL Executable Library any new NM procedures that CM procedures call via Switch
- Incorporate into a CM SL any new CM procedures that NM procedures call via Switch
- Understand the performance factors that affect mixed-mode procedure calls

Mixed-mode Summary and Example

Figures 1-12 and 1-13 summarize mixed-mode procedure calling as provided by Switch:

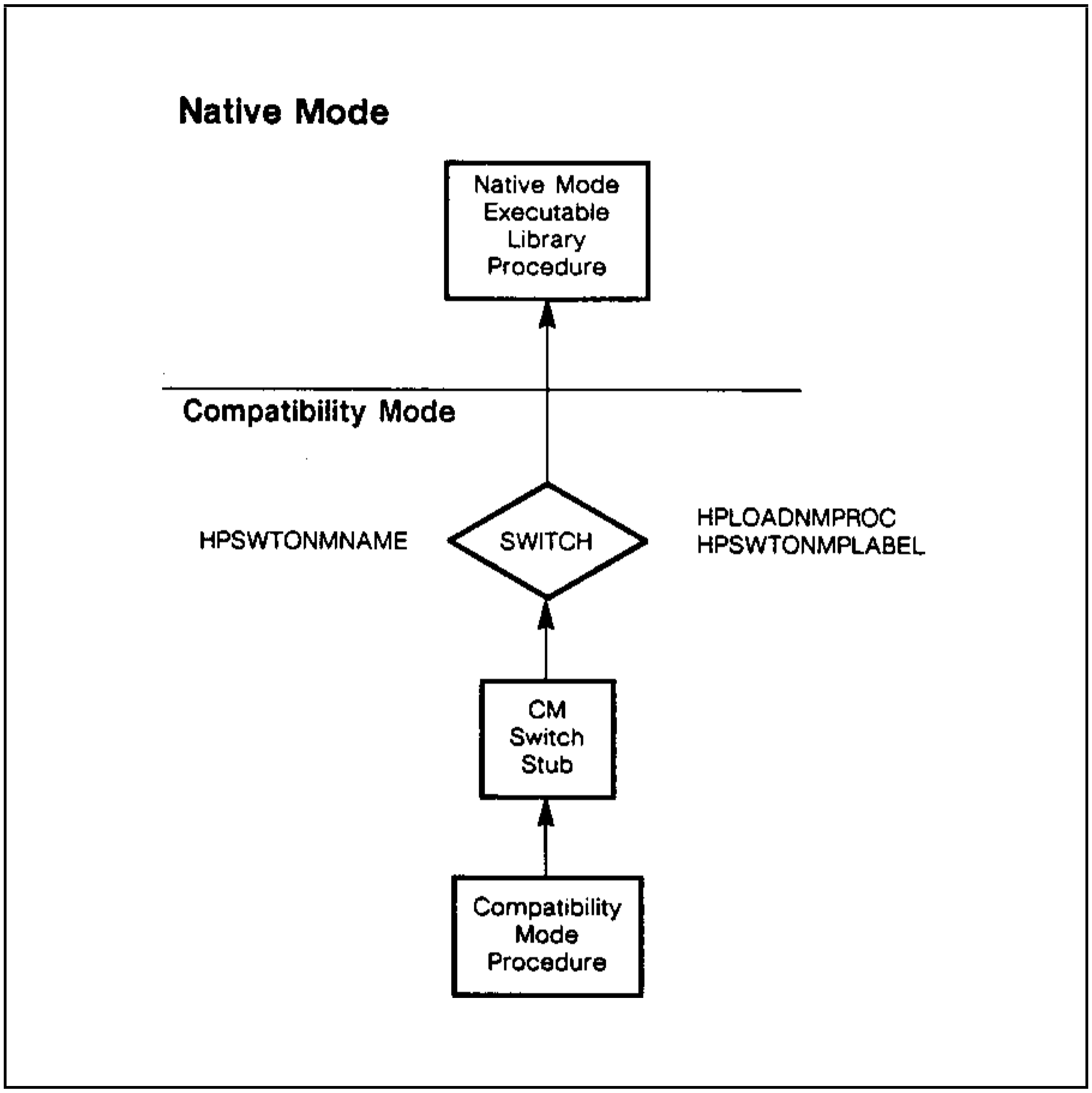


Figure 1-12. CM—> NM Switch Summary

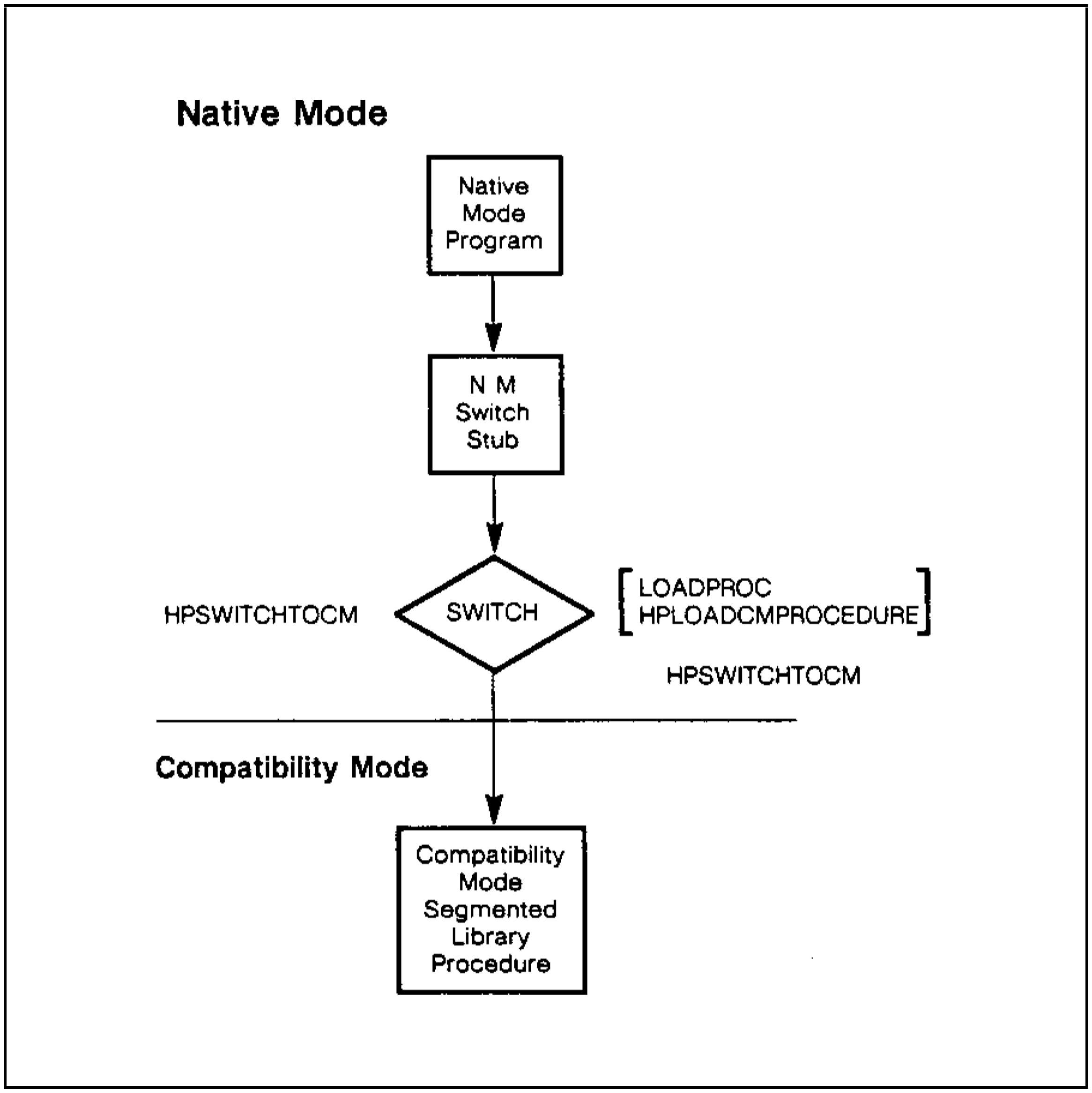


Figure 1-13. NM—> CM Switch Summary

The following series of figures illustrates the migration of an existing COBOL II/V application to Native Mode. You can simply do a STORE/RESTORE to move the application. For even greater performance in Native Mode, you can recompile the source, taking advantage of the NM compilers. However, this application calls an SPL procedure that cannot be migrated to Native Mode. Therefore, you must use Switch to access this SPL procedure (if you do not want to rewrite it in an NM high-level language).

Note In the following figures, shading means that a module is no longer needed.

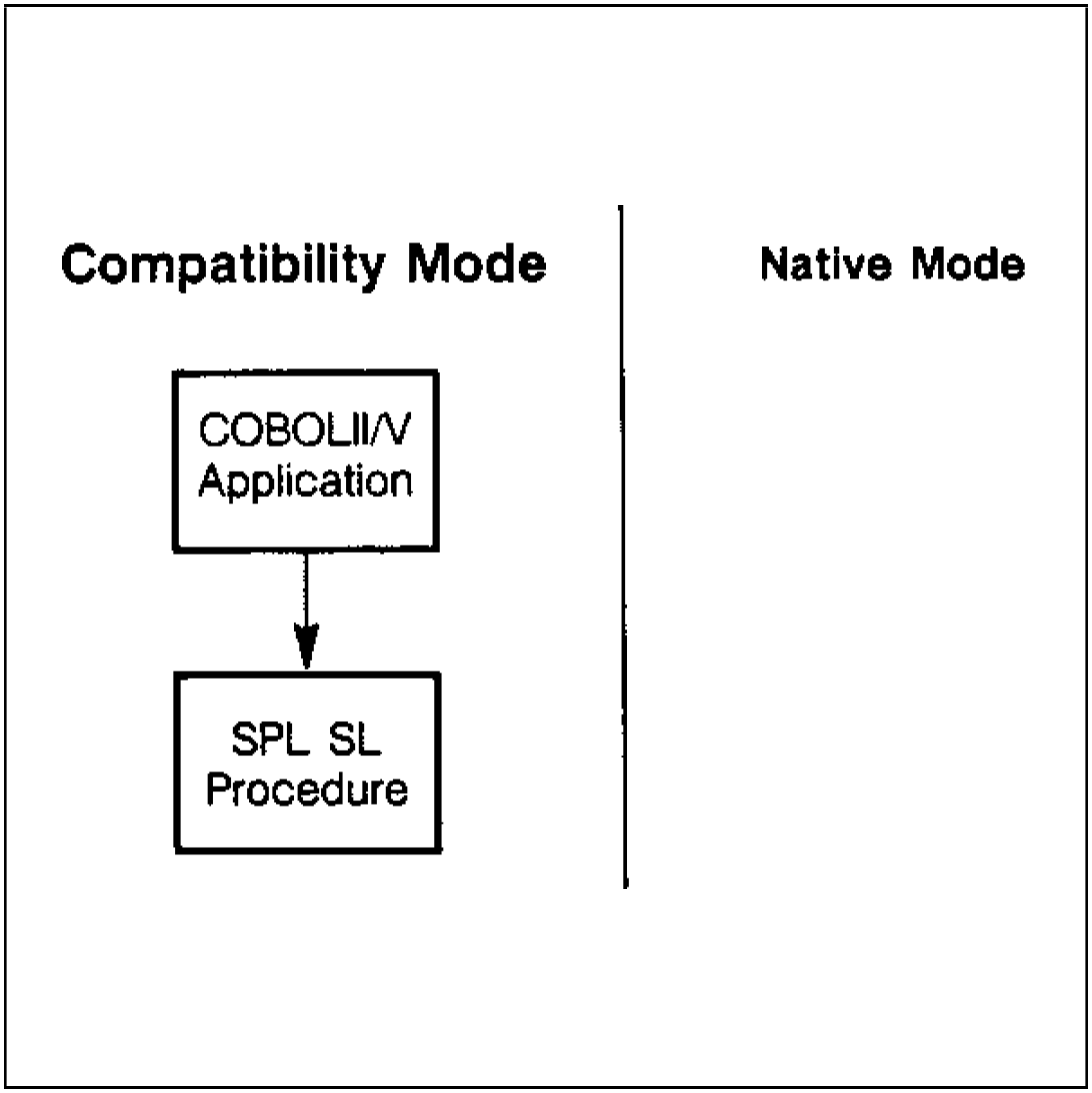


Figure 1-14. Before Migration

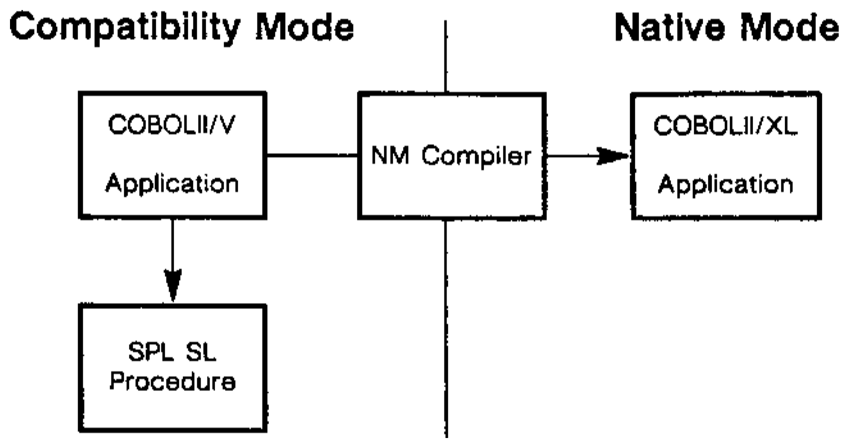


Figure 1-15. Migrating High-level Code to Native Mode

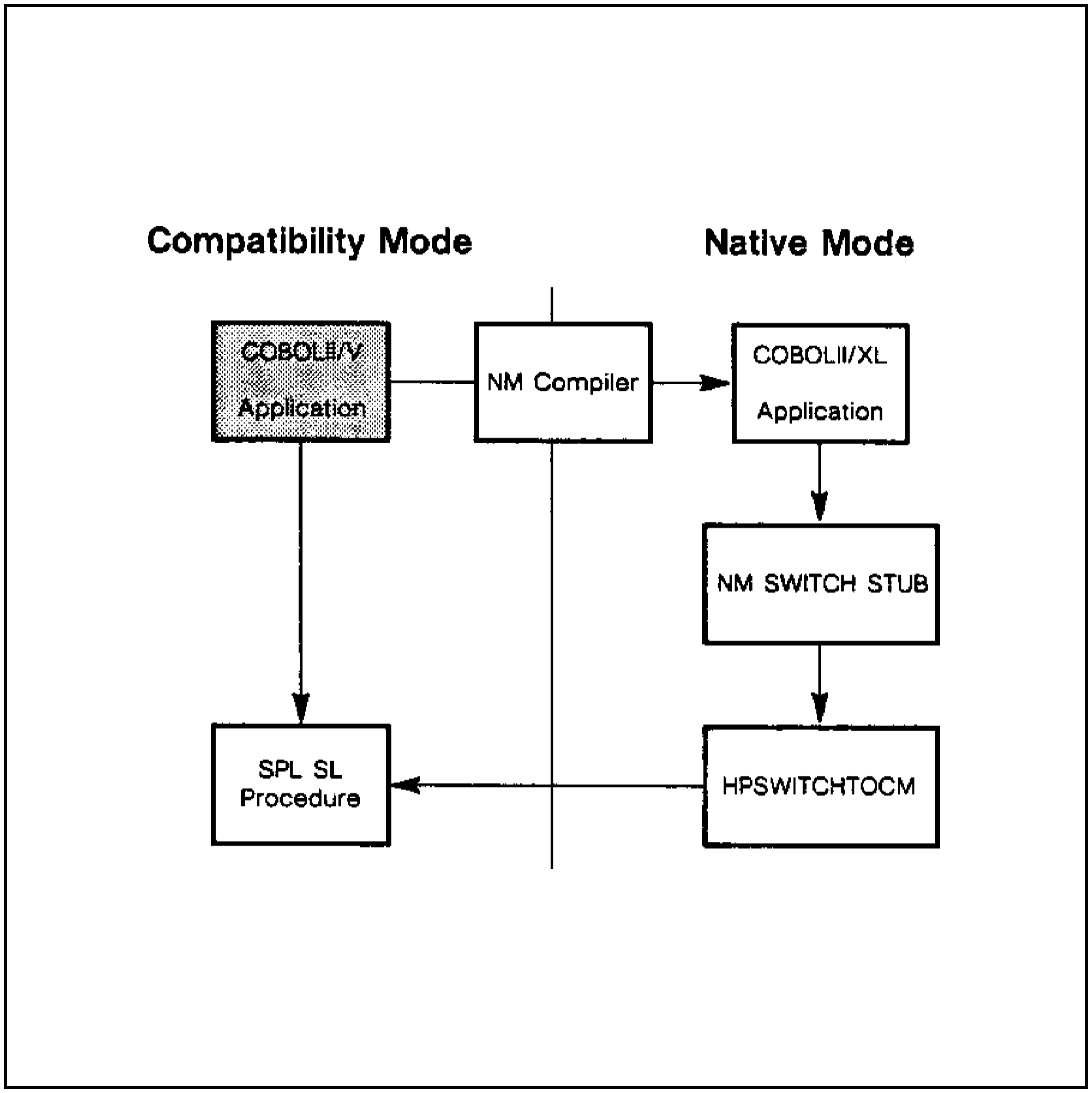


Figure 1-16. Using Switch to Access SPL Procedure

2 Simplifying Switch Programming

Tools have been developed to facilitate the migration process. Among these tools is one that, in most instances, can simplify use of the Switch subsystem (Switch), when switching from Native Mode to Compatibility Mode.

SWitch Assist Tool (SWAT)

The SWitch Assist Tool (SWAT) is an interactive utility that produces the Switch stub source code you can use to make mixed-mode procedure calls in the NM—> CM direction. SWAT prompts you for information regarding the called other-mode procedure and its parameters. In response, SWAT builds source code containing the data structures and logic required to use Switch.

Figure 2-1 illustrates a situation that requires invocation of Switch:

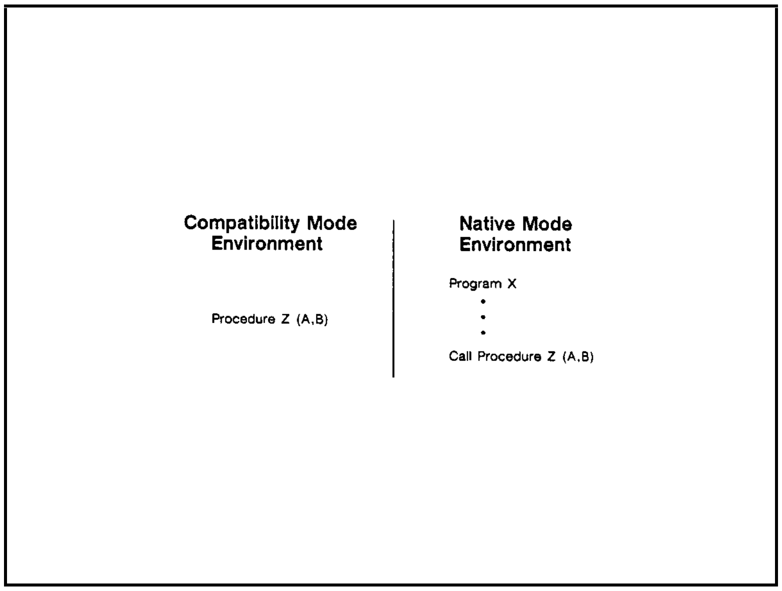


Figure 2-1. Mixed-mode Procedure Call

Procedure Z represents an entity that is not supplied in the Native Mode environment. The SWAT utility can help you handle such unresolved references.

Tools like SWAT enable you to improve performance of migrated applications with a minimum of effort and concern about the architectural differences. You can use SWAT prior to installation to prepare the source code for Switch stubs ahead of time, as well as after the MPE XL-based system is up and running.

Who Uses SWAT?

If you develop applications and subsystems for the HP 3000, then you may be interested in migrating existing applications to MPE XL-based systems. Migrating an application entirely to Native Mode is the preferred approach. However, due to considerations such as the size of the application, the availability of shared procedures, and the language of implementation (SPL, in particular), you may have no choice but to mix modes of execution. The most efficient means for accomplishing this task is to migrate those applications in a phased approach. That requires the mixing of Compatibility Mode and Native Mode code.

Switch provides the ability to make mixed-mode procedure calls. SWAT helps you use Switch to run portions of an application in Compatibility Mode and portions in Native Mode. You will be particularly interested in using SWAT if you have high-level applications that call user-written SPL routines or if your applications call SPL routines for which the source code is not available. Because these two types of routines cannot run in Native Mode, they must either be rewritten entirely or execute in Compatibility Mode. Once you have decided which procedures or modules of your application are to remain in Compatibility Mode, you can use SWAT to generate the necessary Switch stubs to make the switching transparent.

Features

The features of the Switch Assist Tool include the following:

- SWAT's user interface is interactive, flexible, and easy-to-use.
- SWAT generates standardized Switch stub source code in the HP Pascal/XL programming language.
- SWAT facilitates user modification by keeping the source file output in a text file.
- SWAT fully supports most programming options, within the limitations of Switch and MPE XL.
- SWAT automatically appends the appropriate compiler commands to the generated source code.
- SWAT is available upon installation of MPE XL-based systems.
- SWAT standardizes management of the interface between Compatibility Mode and Native Mode.
- SWAT generates the source code required for most possible types of mixed-mode procedure calls in the NM—> CM direction.

Benefits

SWAT's primary benefit is the programmer support it provides by alleviating much of the complexity faced in making mixed-mode procedure calls. It also increases the quality of Switch stub source code by generating standardized source code.

To code a working Switch stub manually, you need the following:

- Knowledge of the stack architecture of MPE V/E-based systems and the register assignments of 900 Series machines
- Familiarity with two system programming languages (SPL and HP Pascal/XL)
- Details of the calling sequences, parameter types, and return values of the target procedures
- Familiarity with the complexities of the Switch intrinsics
- Access to the library management of both CM and NM environments
- Knowledge of how to handle condition codes and status words that are returned when mixed-mode procedure calls are made

By helping you meet these concerns, SWAT helps to make the migration process as smooth and easy as possible.

Furthermore, because much of the information and data structures required by Switch are repetitive in nature and must be duplicated for each parameter passed through Switch, SWAT eliminates much of the tedium otherwise associated with manually coding Switch stubs.

In summary, SWAT serves to increase your productivity and reduce the time required to migrate applications to a Series 900 machine. SWAT benefits those who are migrating application programs and subsystems to MPE XL-based systems in the following ways:

- SWAT incorporates into the logic of the tool the systems and programming knowledge required to produce Switch stub code.
- SWAT eliminates many of the repetitious aspects of stub coding.
- SWAT helps assure the correct execution of mixed-mode procedure calls by generating standardized Switch stub source code as its output.

- For those who prefer to code Switch stubs manually, or in some instances must do so, SWAT provides examples of good coding practices and can act as a template for advanced stubs.

Switch Stub Operation

Figure 2-2 demonstrates how Switch stubs and the Switch subsystem fit into the overall flow of program execution:

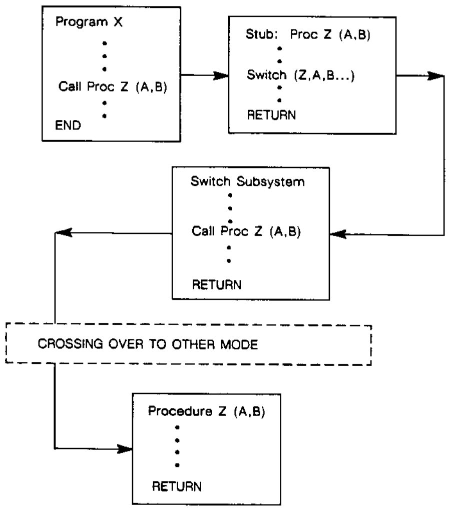


Figure 2-2. Switch Scenario

The Switch stub acts as an interceptor/intermediary that calls the Switch subsystem and passes it the parameters of the called procedure and any other information required by Switch. Switch itself calls the other-mode procedure.

Generating Switch Stubs Automatically

The Switch Assist Tool and its user interface help you create the Switch stub source code, which has the following components:

- Compiler commands
- Data structures
- Logic

Specifically, in producing the source code of a stub procedure, SWAT does the following:

1. Converts the SPL procedure name and variable names to valid HP Pascal/XL names
2. Generates the top global area code
3. Generates the stub procedure declaration and parameters
4. Generates the stub procedure local variables and its `BEGIN` statement
5. Generates the stub procedure body code to prepare the Switch call parameters
6. Generates the code to call the Switch intrinsic
7. Generates the code to check the status and calls the `QUIT` intrinsic when Switch encounters an error
8. Generates the `END` statement of the stub procedure
9. Generates the bottom global area code

Note SWAT output is a stub written in HP Pascal/XL, no matter what language the original application was written in. The Switch stub source code produced by SWAT is, in turn, input for the HP Pascal/XL compiler, which produces Switch stub object code. For more information on how to invoke this compiler, see the discussion of the `PASXL` command in the *MPE XL Commands Reference Manual* (32650-90003). Compilation of that source must be on an MPE XL-based machine. The resulting object code is input to the MPE XL Link Editor, where you can either install the object code in an NM Executable Library or link it into an NM program file. For more information on the Link Editor, refer to *Link Editor/XL Reference Manual* (32650-90030).

Invoking SWAT

To invoke SWAT, enter the following at the system prompt:

```
RUN SWAT.PUB.SYS
```

What SWAT Needs to Know

In order to generate NM—> CM Switch stubs, SWAT needs to know the following information:

- Name of the text file into which SWAT should place the generated source code
- Name of the procedure to be called
- Location of that procedure (for example, CM system, account, or group Segmented Library)
- Type (expressed in SPL terminology) of the function return value if the Switch call is to a function
- Whether the stub procedure should pass any condition code to the caller
- Number of parameters Switch is to pass to the procedure
- Addressing method, data type, I/O type, and length of each parameter in the calling sequence of the procedure Switch is to invoke (for arrays, length is specified as the number of elements)

Note The process of gathering information about parameters is iterative.

How SWAT Obtains Information

When you run SWAT, SWAT uses VPLUS screens to prompt you for this required information. There are a total of seven screens that you may encounter as you enter the information required by SWAT, request help on how to complete a form screen, edit forms or confirm your satisfaction with the completed forms, and await the result of the stub generation process. The names of the screens (discussed in detail below) are as follows:

- MAIN menu screen (see Figure 2-3)
- PROCINFO menu screen (see Figure 2-4)
- PARMINFO menu screen (see Figure 2-5)
- ARRAYLEN menu screen (see Figure 2-6)
- HELP facility screens (see Figure 2-7 and Figure 2-10)
- COMMIT screen (see Figure 2-11)
- PROGRESS MESSAGES screen (see Figure 2-12)

Typical SWAT Work Session

The normal progression in a work session with SWAT is to complete the MAIN menu information first and then the PROCINFO menu. Based on the information you entered in the MAIN menu, the PARMINFO menu appears repeatedly, once for every parameter you entered by name in the MAIN menu. If, in the course of filling out the PARMINFO menu, you indicate that a parameter is an array, the ARRAYLEN menu will appear.

The contents of a HELP screen depend on the menu you are in at the time of the request for help.

After committing your menu selections via the COMMIT screen, stub generation begins and you are placed in the PROGRESS MESSAGES screen so you can track the generation process.

After generation of Switch stub source code is complete, the MAIN menu screen reappears, allowing you to build another stub or to exit SWAT and return to the system Command Interpreter.

Inputting Information

You move from field to field and item to item within a menu by means of the Cursor Control keys or the Tab key. To complete some fields, you must enter a name or numerical value. In other instances, the field consists of a list of options. You complete such fields by placing a nonblank character next to the option of your choice.

At any point before actually committing the information in the menus and beginning the generation of a stub, you can move back and forth through the menus in order to change their contents or request help on how to complete a menu item.

To input the information entered on the menu to SWAT, press the (Enter) key. This records your selections, but does not prevent you from reviewing and editing them. When you press the (Enter) key, SWAT performs error-checking on your selections.

Resolving Errors

The Switch Assist Tool helps you resolve any problems that may arise in completing a menu.

The tool uses highlighting to indicate any fields that contain errors. It also prints a diagnostic message at the bottom of the screen. You can use the highlighting and the message to make corrections to the menu.

SWAT places the cursor in the topmost field containing an error. After making the correction suggested by the message, press (Enter). If your correction resolves the error, both the message and the highlighting disappear. If not, the highlighting remains, and a new message may appear to help you resolve the problem.

If your menu contains more than one error, you can resolve them one by one in this same manner. After you resolve the topmost field, the highlighting of other errors remains, the cursor moves to the next field containing an error, and a message appropriate to that error appears at the bottom of the screen. In this way, the tool guides you through the correction process.

When the last error is resolved, the Switch Assist Tool records your entries when you press (Enter) and brings up the next menu.

MAIN Menu Screen

The MAIN menu screen is the first to appear upon invoking SWAT. To complete this menu, you need to supply the following information:

- Name of the file to hold the source code generated by SWAT
- Name of the target procedure
- Names of the target's parameters (up to 32)

Designating the Source Code File

The file name to hold the source code generated by SWAT can have a maximum of eight characters. SWAT creates a fixed ASCII type file in the group and account in which you are currently logged on. If you do not enter a file name, you receive a message indicating that you cannot leave this field blank. The cursor is repositioned to the file name field.

Note Fully qualified file names and passwords are not allowed.

Designating the Target CM Procedure

The target procedure is the procedure to be called by means of the SWAT-generated stub. Its name can have a maximum of sixteen characters. Again, if you do not enter a name, you receive a message indicating that it is not possible to leave this field blank. The cursor returns to the beginning of this field.

Designating the Target Procedure Parameters

SWAT allows calls to target procedures with up to 32 parameters. It is important to note that you must enter target parameters in the same order as the calling sequence of the target procedure. Parameter names can be up to a maximum of 32 characters long. You may leave these fields blank, indicating that the target procedure has no parameters.

Note Procedure and parameter names are not localizable. They are limited to the alphanumeric characters of the English alphabet. All entries are checked for validity. However, the Switch Assist Tool does not check for use of HP Pascal/XL reserved words and predefined routines as names. For more information about reserved words and predefined routine names in HP Pascal/XL, refer to the *HP Pascal/XL Reference Manual* (31502-90001).

Refer to Figure 2-3 for an illustration of the MAIN menu screen.

Figure F02-03 here.

Figure 2-3. MAIN Menu Screen

MAIN Menu Function Keys

There are several function key options available to you on the MAIN menu screen:

- Press F4 to refresh the screen.
- Press F6 to proceed to the next menu (used for reviewing and editing menu contents).

- Press F7 to view the HELP screen on the MAIN menu.
- Press F8 to exit SWAT and return to the system Command Interpreter.

Caution Pressing F8 at any time returns you immediately to the system prompt, causing you to lose any information entered up to that point.

Completing the MAIN Menu Screen

To input the information entered on the menu to SWAT and initiate error-checking, press the (Enter) key.

The process of correcting MAIN menu errors is the same as that described in "Resolving Errors".

PROCINFO Menu Screen

The PROCINFO menu screen collects information concerning the procedure to be invoked by the SWAT-generated stub. The name of this procedure is displayed at the top of the PROCINFO menu. It is the same name that you entered on the MAIN menu screen. You can change that name on the PROCINFO screen if you desire. Switch needs to know the following information about this procedure:

- Location of the target procedure (CM system, account, or group Segmented Library)
- Data type of its return value if it is a function
- Privilege level of the target procedure
- Whether the target procedure returns a condition code

For each of the five fields listed above, the appropriate number of legal options is provided on the PROCINFO menu. You can select one value for each field. You make your selection by placing a nonblank character in the space provided next to the appropriate value. For some fields, default values are provided.

You cannot leave any of the five fields blank. If you do so, the cursor is repositioned at the blank field, and an appropriate error message is sent to the screen.

Designating the Location of the Target Procedure

The target procedure must be located in some CM Segmented Library (SL). This may be the system library, an account library, or a group library. If you specify the system library, only the system SL is searched. If you specify an account library, the account library is searched first, then the system library. Finally, if you specify a group library, the group library is searched first, followed in turn by that group's account library and the system library. The target procedure need not be installed in order for SWAT to generate Switch stub source code for calling the target.

Designating the Function Return Type of the Target Procedure

If the target procedure is not a function (that is, does not return a value as a functional return value), you should select the `NONE` value in the field designating the function return type. Otherwise, select the appropriate data type from the list of possibilities.

Designating the Privilege Level of the Target Procedure

SWAT designates `NON-PRIVILEGED` as the default value for the privilege level of the target procedure. To access user-written CM SL procedures, simply accept this default value.

Designating Whether the Target Procedure Returns a Condition Code

Every process has an associated condition code. A condition code is a temporary value that provides information about process execution. From the condition code value, you can learn some basic information regarding what happened during execution of the process. This code has three possible values, which are assigned as follows:

- CCE, or Condition Code Equal (=), generally indicating that the process executed successfully
- CCG, or Condition Code Greater Than (>), indicating that a special condition occurred, but may not have affected the execution of the process
- CCL, or Condition Code Less Than (<), indicating that the process did not execute successfully, but the error condition may be recoverable

Since this code is affected by many processes, you should check its value immediately upon return from a process. To return the condition code from the target CM routine, select the `YES, RETURN CONDITION CODE` option on the PROCINFO screen.

Figure 2-4 provides an example of the PROCINFO menu screen.

re F02-04 here.

Figure 2-4. PROCINFO Menu Screen

PROCINFO Menu Function Keys

There are several function key options available to you on the PROCINFO menu screen:

- Press F1 to return to the MAIN menu.
- Press F4 to refresh the screen.
- Press F5 to go back to the previous menu.
- Press F6 to proceed to the next menu (for reviewing/editing information, not recording it).
- Press F7 to view the HELP screen on the PROCINFO menu.
- Press F8 to exit SWAT and return to the system Command Interpreter.

Completing the PROCINFO Menu

To input the information entered on the menu to SWAT and initiate error checking, press the (Enter) key.

The process of correcting PROCINFO menu errors is the same as that described in "Resolving Errors".

PARMINFO Menu Screen

The PARMINFO menu screens collect information concerning the parameters to be passed to the target routine. This screen appears repeatedly, once for each parameter you entered on the MAIN menu screen. The name of each parameter appears at the top of the PARMINFO menu screen, just as you entered it on the MAIN menu. You can change that name on the PARMINFO screen, if desired. The order in which the PARMINFO screens appear reflects the order in which you entered these parameter names and, thus, the call sequence of the target routine.

The type of information about each parameter that Switch needs includes the following:

- Addressing method of the parameter
- Data type of the parameter
- I/O type of the parameter, relative to the target CM procedure

- Whether the parameter is an array

The appropriate legal options are provided on the PARMINFO menu. You can select one value for each field. You make your selection by placing a nonblank character in the space provided next to the appropriate value.

You cannot leave any of the fields blank. If you do so, the cursor is repositioned at the blank field. SWAT also checks that you have not specified a multi-element item as a value parameter.

Designating a Parameter's Addressing Method

In specifying the addressing method of a parameter, you indicate whether the parameter is passed as a reference parameter or as a value parameter.

Note In COBOL, all parameters are passed by reference unless surrounded by \\.

Designating a Parameter's Data Type

The data type of a parameter must be specified in SPL terminology. SWAT does not handle the following data types: pointers, labels, plabels, and arrays passed by value. Any types not covered require manual programming of Switch stubs. The section entitled "Special Cases" provides more detail about unsupported data types.

Note For further details on SPL addressing methods and data types, refer to *Systems Programming Language Reference Manual* (30000-90024).

Designating a Parameter's I/O Type

In the field designating the I/O type, you designate whether the parameter is an input parameter, an output parameter, or an input/output parameter, relative to the target CM procedure. If the parameter is an input parameter, data goes from the NM to the CM stack, but not back. On the other hand, for an output parameter, data is returned from the CM stack to the NM stack. In the case of input/output parameters, data goes from the NM to the CM stack, and back again.

Designating Whether a Parameter Is an Array

You must indicate whether a parameter is an array. You do so by designating the appropriate option in the `ARRAY SPECIFICATION` field. If you indicate that a parameter is an array, the ARRAYLEN menu screen subsequently appears to collect information about the length of the array parameter.

A sample PARMINFO menu screen appears in Figure 2-5.

Figure F02-05 here.

Figure 2-5. PARMINFO Menu Screen

PARMINFO Menu Function Keys

There are several function key options available to you on the PARMINFO menu screen:

- Press F1 to return to the MAIN menu.
- Press F4 to refresh the screen.
- Press F5 to go back to the previous screen.
- Press F6 to proceed to the next screen (for reviewing/editing information, not recording it).
- Press F7 to view the HELP screen on the PARMINFO menu.
- Press F8 to exit SWAT and return to the system Command Interpreter.

Completing the PARMINFO Menu

To input the information entered on the menu to SWAT and initiate error checking, press the (Enter) key.

The process of correcting PARMINFO menu errors is the same as that described in "Resolving Errors".

ARRAYLEN Menu Screen

The ARRAYLEN menu screen appears only if you have indicated that a parameter is an array on the PARMINFO menu screen. The ARRAYLEN screen collects information concerning array parameters to be passed to the routine you want to call in the Compatibility Mode environment.

The array information required by the Switch Assist Tool includes the following:

- Length of the array parameter
- How the length value should be interpreted

You cannot leave any of the fields blank. If you do so, the cursor is repositioned at the blank field.

Designating the Length Value of an Array Parameter

You have two options when specifying the length (size) of an array parameter. One way is to designate a specific constant value in the range of 1 to 65,535. If that is your choice, enter a decimal number in the space provided. The number of array elements allowed depends on the SPL data type in the target CM array parameter. Table 2-1 lists the allowed array element ranges for the various SPL data types.

Table 2-1. Number of Allowed Array Elements By Type

SPL Array Element Type	No. of Array Elements Allowed
Byte	1 ... 65,535
Integer	1 ... 32,767
Logical	1 ... 32,767
Double	1 ... 16,383
Real	1 ... 16,383
Long	1 ... 8,191

You can also use the value of a parameter to specify the length of an array. If that is your choice, enter the parameter name in the space provided. The name specified must be the name of another parameter being passed to the target procedure. If it is not, SWAT displays an error message.

Designating How to Interpret the Array Length Usage Value

In addition to indicating the length of an array (either as a constant value or by means of a parameter name), you must also indicate how SWAT should interpret that value. The length usage value specifies one of the following:

- Number of array elements.
- Number of bytes in the array.
- Negative = Bytes / Positive = Elements Rule

If you select the third option, the following apply:

- If the parameter has a positive value, that value represents the size as the number of elements.
- If the value of the parameter is negative, the parameter represents the size as the number of bytes.

Select the appropriate option by placing a nonblank character in the space provided next to your choice.

Caution If you specify an array size larger than its declared size, unpredictable results occur when the stub is executed.

A sample ARRAYLEN menu screen appears in Figure 2-6.

Figure F02-06 here.

Figure 2-6. ARRAYLEN Menu Screen

ARRAYLEN Menu Function Keys

There are several function key options available to you on the ARRAYLEN menu screen:

- Press F1 to return to the MAIN menu.
- Press F4 to refresh the screen.
- Press F5 to go back to the previous screen.
- Press F6 to proceed to the next screen (for reviewing/editing information, not recording it).
- Press F7 to view the HELP screen on the ARRAYLEN menu.
- Press F8 to exit SWAT and return to the system Command Interpreter.

Completing the ARRAYLEN Menu

To input the information entered on the menu to SWAT and initiate error checking, press the (Enter) key.

The process of correcting ARRAYLEN menu errors is the same as that described in “Resolving Errors”.

HELP Screens

There are HELP screens for each of the menu screens that make up the SWAT user interface.

There are several function key options available to you in the HELP facility:

- Press F4 to refresh the screen.
- Press F5 to go back to the previous screen.
- Press F6 to proceed to the next screen.
- Press F8 to exit the HELP facility.

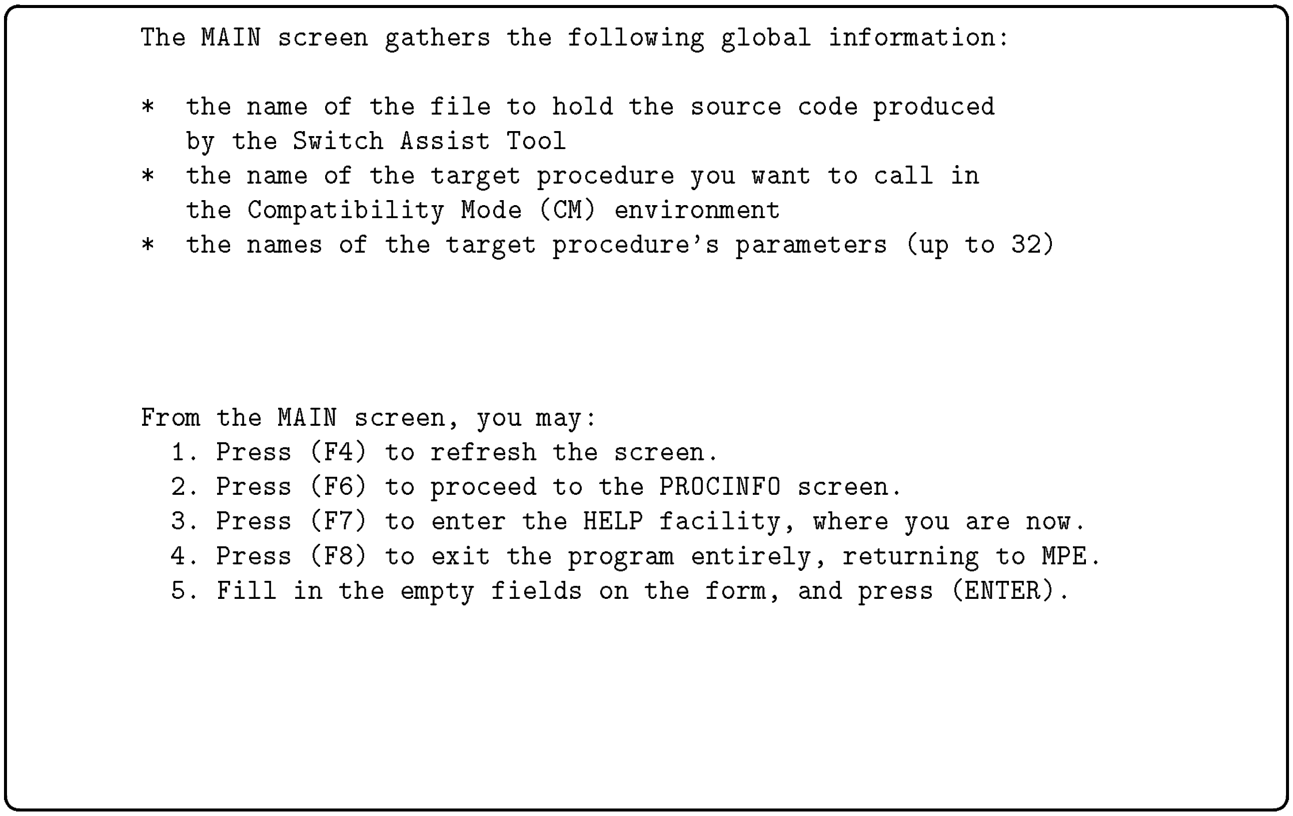
MAIN Menu HELP Screen

The MAIN menu HELP screen provides information to help you complete the items requested in the MAIN menu screen:

- Name of the file to hold source code generated by SWAT

- Name of the target procedure
- Name(s) of the parameters of the target procedure (up to 32)

Figure 2-7 illustrates the appearance of the introductory MAIN menu HELP screen.



```
The MAIN screen gathers the following global information:

*  the name of the file to hold the source code produced
   by the Switch Assist Tool
*  the name of the target procedure you want to call in
   the Compatibility Mode (CM) environment
*  the names of the target procedure's parameters (up to 32)




From the MAIN screen, you may:
  1. Press (F4) to refresh the screen.
  2. Press (F6) to proceed to the PROCINFO screen.
  3. Press (F7) to enter the HELP facility, where you are now.
  4. Press (F8) to exit the program entirely, returning to MPE.
  5. Fill in the empty fields on the form, and press (ENTER).
```

Figure 2-7. Main Menu Help Screen

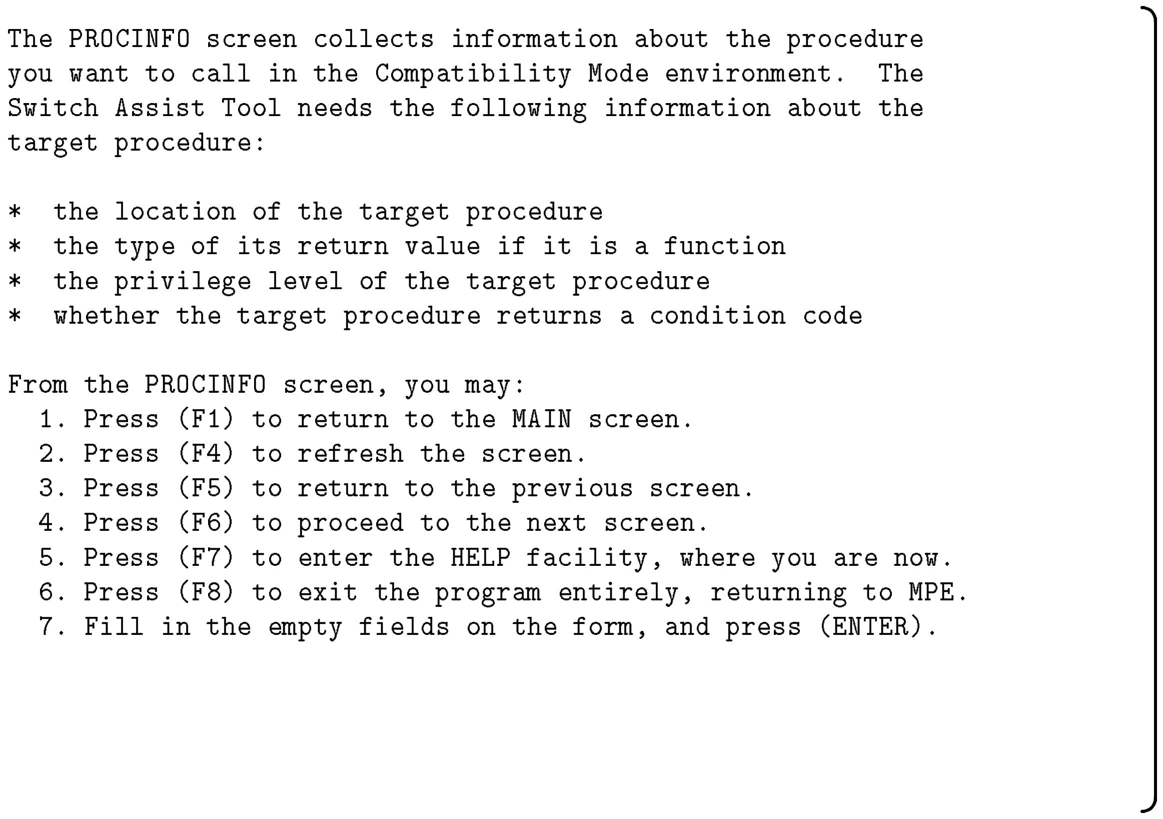
PROCINFO Menu HELP Screen

The PROCINFO menu HELP screen provides information to help you complete the items requested in the PROCINFO menu screen:

- Location of the target procedure
- Type of the return value if the target procedure is a function

- Privilege level of the target procedure
- Whether the target procedure returns a condition code

Figure 2-8 illustrates the appearance of the introductory PROCINFO menu HELP screen.



```
The PROCINFO screen collects information about the procedure
you want to call in the Compatibility Mode environment.  The
Switch Assist Tool needs the following information about the
target procedure:

*  the location of the target procedure
*  the type of its return value if it is a function
*  the privilege level of the target procedure
*  whether the target procedure returns a condition code

From the PROCINFO screen, you may:
  1. Press (F1) to return to the MAIN screen.
  2. Press (F4) to refresh the screen.
  3. Press (F5) to return to the previous screen.
  4. Press (F6) to proceed to the next screen.
  5. Press (F7) to enter the HELP facility, where you are now.
  6. Press (F8) to exit the program entirely, returning to MPE.
  7. Fill in the empty fields on the form, and press (ENTER).
```

Figure 2-8. PROCINFO Menu Help Screen

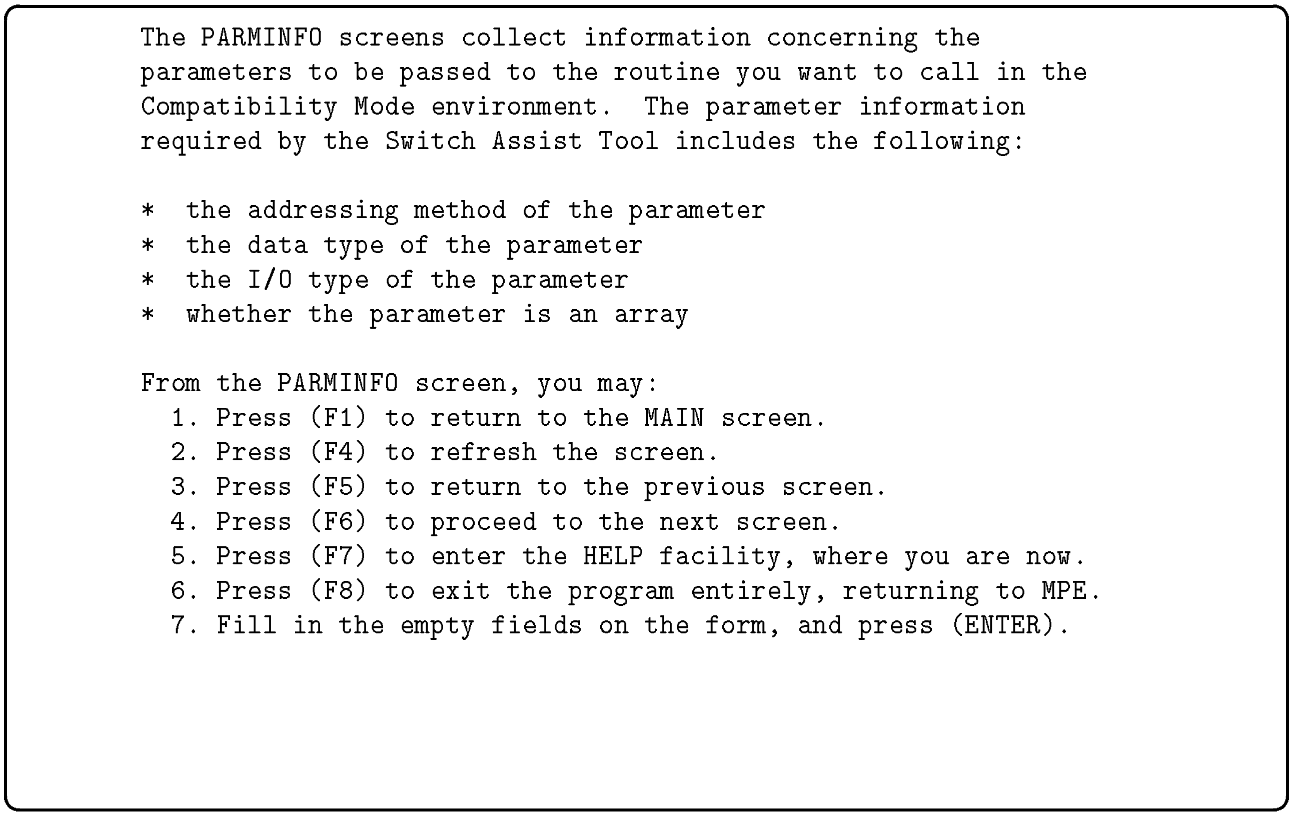
PARMINFO Menu HELP Screen

The PARMINFO menu HELP screen provides information to help you complete the items requested in the PARMINFO menu screen:

- Addressing method of a parameter
- Data type of a parameter

- I/O type of a parameter
- Whether the parameter is an array and, if so, how many elements are in that structure

Figure 2-9 illustrates the appearance of the introductory PARMINFO menu HELP screen.



```
The PARMINFO screens collect information concerning the
parameters to be passed to the routine you want to call in the
Compatibility Mode environment.  The parameter information
required by the Switch Assist Tool includes the following:

*  the addressing method of the parameter
*  the data type of the parameter
*  the I/O type of the parameter
*  whether the parameter is an array

From the PARMINFO screen, you may:
  1. Press (F1) to return to the MAIN screen.
  2. Press (F4) to refresh the screen.
  3. Press (F5) to return to the previous screen.
  4. Press (F6) to proceed to the next screen.
  5. Press (F7) to enter the HELP facility, where you are now.
  6. Press (F8) to exit the program entirely, returning to MPE.
  7. Fill in the empty fields on the form, and press (ENTER).
```

Figure 2-9. PARMINFO Menu Help Screen

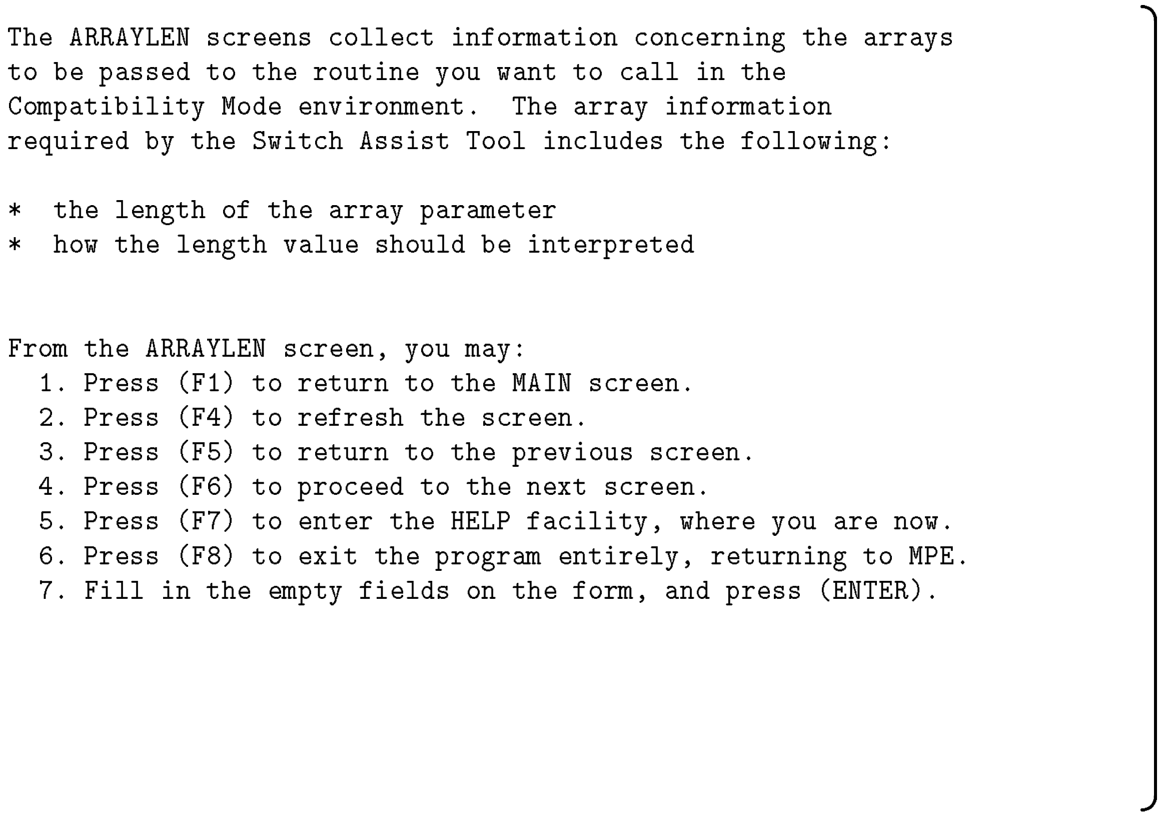
ARRAYLEN Menu HELP Screen

The ARRAYLEN menu HELP screen provides information to help you complete the items requested in the ARRAYLEN menu screen:

- Length of the array parameter

- How the array length value (either a constant value or a parameter name) should be interpreted

Figure 2-10 illustrates the appearance of the introductory ARRAYLEN menu HELP screen.



```
The ARRAYLEN screens collect information concerning the arrays
to be passed to the routine you want to call in the
Compatibility Mode environment.  The array information
required by the Switch Assist Tool includes the following:

*  the length of the array parameter
*  how the length value should be interpreted


From the ARRAYLEN screen, you may:
  1. Press (F1) to return to the MAIN screen.
  2. Press (F4) to refresh the screen.
  3. Press (F5) to return to the previous screen.
  4. Press (F6) to proceed to the next screen.
  5. Press (F7) to enter the HELP facility, where you are now.
  6. Press (F8) to exit the program entirely, returning to MPE.
  7. Fill in the empty fields on the form, and press (ENTER).
```

Figure 2-10. ARRAYLEN Menu Help Screen

COMMIT Screen

The COMMIT screen does not request that you supply information to complete a menu. Instead, this screen presents you with the following function key options:

- Press F1 to return to the MAIN menu.

- Press F2 to commit your menu selections and allow SWAT to begin stub generation.
- Press F4 to refresh the screen.
- Press F5 to return to the previous screen.
- Press F7 to view the HELP screen that deals with the COMMIT screen.
- Press F8 to leave SWAT.

These options let you decide whether SWAT should begin generating Switch stub source code or whether you need to edit information entered in the menu screens. Recall that the Previous (and Next) function keys allow you to review and edit any SWAT menu screen. The COMMIT screen does not allow you to begin Switch stub generation if any menus remain unresolved. The COMMIT screen presents messages that indicate which menus or items need resolution.

Figure 2-11 illustrates the appearance of the COMMIT screen:

Figure F02-11 here.

Figure 2-11. COMMIT Screen

PROGRESS MESSAGES Screen

The PROGRESS MESSAGES screen appears after you have committed your menu selections by means of the COMMIT screen. In addition to displaying the name of the stub procedure and the line number currently being processed, the PROGRESS MESSAGES screen has space to display any progress, result, or error messages that SWAT has to report as it generates the source code for the stub. An example PROGRESS MESSAGES screen follows in Figure 2-12:

Figure F02-12 here.

Figure 2-12. PROGRESS MESSAGES Screen

Completion of Stub Generation

After generation of Switch stub source code is complete, the MAIN menu screen reappears, allowing you to build another stub or to exit SWAT and return to the system Command Interpreter.

Modifying SWAT Output

The output file generated by SWAT contains the compiler statements and external declarations necessary to permit compilation of the stub procedure source code as a single entity. However, for a variety of reasons, you may wish to modify this file. Because SWAT generates source code in a text file, you can use existing HP 3000 editors to modify the source produced by the tool.

Reasons for modifying SWAT output include the following:

- Adding additional compiler commands, error-handling statements, and/or comments
- Merging several Switch stub source modules into one file (SWAT produces only one stub per file)
- Making changes to handle the specific situations for which SWAT cannot generate complete stubs (see "Special Cases")

To merge Switch stubs, you need to identify the global area of a stub. The top global area consists of the first line through the comment line designating the end of the outer block declarations. The bottom global area begins following the line containing the comment that designates the end of the stub procedure. If you combine stubs, only one such global area is necessary. You can delete the global areas from all but one stub, and then insert any number of stub procedures between those global declarations. Refer to Appendixes D and E for use files that automate the process of merging several Switch stub source modules into one file.

Using SWAT Output

SWAT output is a stub written in HP Pascal/XL, no matter what language the original application was written in. The Switch stub source code produced by SWAT is, in turn, input for the HP Pascal/XL compiler, which produces Switch stub object code. The following command would compile a SWAT-generated stub named `mystub`, yielding an object code file named `myobj`:

```
        PASXL mystub, myobj
```

For more information, refer to the discussion of the `PASXL` command in the *MPE XL Commands Reference Manual* (32650-90003). Compilation of that source must be on an MPE XL-based machine.

The resulting object code is input to the MPE XL Link Editor, where you can either install the object code in an NM Executable Library for shared use or link it into an NM program file. Example 3-10 of this manual illustrates a sample LinkEdit session. For more information on the Link Editor, refer to the *Link Editor/XL Reference Manual* (32650-90030).

SWAT Requirements

The Switch Assist Tool is designed to run on MPE XL-based systems. SWAT has certain hardware and software requirements.

Hardware Requirements

SWAT requires the minimum system configuration listed below for operation on MPE XL-based systems:

- 1 900 Series System Processing Unit
- 1 System disc
- 1 System printer
- 1 System tape drive
- 1 Block mode terminal (as supported by VPLUS/XL)

Software Requirements

In addition to the SWAT program file (`SWAT.PUB.SYS`), the tool requires the following additional system software:

- VPLUS/XL fast form file `SWATFORM` (part of the tool)
- VPLUS/XL subsystem
- Pascal run-time library
- Message catalog file `SWATCAT`
- NLS message catalog subsystem

Note All of the above are part of the Fundamental Operating Software (FOS).

To compile the source code generated by SWAT, you must have the HP Pascal/XL compiler.

Example Source File Generated by SWAT

The following is an example of the source code generated by SWAT. You can use this source to call an SPL function `FREAD` with three parameters `FILENUMBER`, `TARGET`, and `TCOUNT`.

Note This FREAD stub is only an example. The `FREAD` intrinsic is directly accessible from both CM and NM, without the need of writing a stub to gain access.

```
$subprogram$
$check_actual_parm 0$
$check_formal_parm 0$
$os 'MPE/XL'$
$standard_level 'ext_modcal'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$
{***********************************************}
{*                                             *}
{* Generated: WED, OCT 21, 1987, 10:21 AM      *}
{*                                             *}
{* Switch Assist Tool HP30363A.00.00           *}
{*                                             *}
{***********************************************}

PROGRAM Hp_stub_outer_block(input, output);
```

```
{This program is an example program written by the   }
(SWitch Assist Tool.                                 }

CONST
   Hp_Pidt_Known  = 0;  {by number}
   Hp_Pidt_Name   = 1;  {by name}
   Hp_Pidt_Plabel = 2;  {by PLABEL}

   Hp_System_SL        = 0;  {System SL}
   Hp_Logon_Pub_SL     = 1;  {Logon PUB SL}
   Hp_Logon_Group_SL   = 2;  {Logon GROUP SL}
   Hp_Pub_SL           = 3;  {Program's PUB SL}
   Hp_Group_SL         = 4;  {Program's GROUP SL}

   Hp_Method_Normal  = 0; {Not callable from split stack}
   Hp_Method_Split   = 1; {Callable in split stack mode}
   Hp_Method_No_Copy = 2;

   Hp_Parm_Value      = 0; {value parameter}
   Hp_Parm_Word_Ref   = 1; {reference parm, word addr}
   Hp_Parm_Byte_Ref   = 2; {reference parm, byte addr}

   Hp_Ccg      = 0; {condition code greater (>)}
   Hp_Ccl      = 1; {condition code less (<)}
   Hp_Cce      = 2; {condition code equal (=)}
   Hp_All_Ok   = 0; {Used in status check}

TYPE
   Hp_BIT8     = 0..255;
   Hp_BIT16    = 0..65535;
   Hp_BIT8_A1  = $ALIGNMENT 1$ Hp_BIT8;
   Hp_BIT16_A1 = $ALIGNMENT 1$ Hp_BIT16;

   Hp_SCM_PROCEDURE = PACKED RECORD
   Hp_CM_PROC_NAME = PACKED ARRAY[1..16] OF CHAR;

   Hp_GENERIC_BUFFER = PACKED ARRAY[1..65535] OF CHAR;
      CASE Hp_p_proc_id_type : Hp_BIT8 OF
```

```
         Hp_Pidt_Known : (Hp_p_fill     :  Hp_BIT8_A1;
                          Hp_p_proc_id  :  Hp_BIT16_A1
                      );
         Hp_Pidt_Name : (Hp_p_lib       : Hp_BIT8_A1;
                         Hp_p_proc_name : Hp_CM_PROC_NAME
                     );
         Hp_Pidt_Plabel: (Hp_p_plabel : Hp_BIT16_A1);
      END; {record}

   Hp_SCM_IO_TYPE = SET OF (Hp_input, Hp_output);

   Hp_PARM_DESC = PACKED RECORD
      Hp_pd_parmptr : GLOBALANYPTR;
      Hp_pd_parmlen : Hp_BIT16;
      Hp_pd_parmtype : Hp_BIT16;
      Hp_pd_io_type : Hp_SCM_IO_TYPE;
   END;  {record}

   Hp_SCM_PARM_DESC_ARRAY = ARRAY[0..31] OF Hp_PARM_DESC;

   HP_STATUS_TYPE   =  RECORD
      CASE INTEGER OF
         0 : (Hp_all : INTEGER);
         1 : (Hp_info : SHORTINT;
              Hp_subsys : SHORTINT);
   END; {record}

{Declare all types which can be passed to this stub   }
{so that 16 bit alignments are allowed.               }

HP_SHORTINT   =   $ALIGNMENT 2$ SHORTINT;
HP_INTEGER    =   $ALIGNMENT 2$ INTEGER;
HP_REAL       =   $ALIGNMENT 2$ REAL;
HP_LONG       =   $ALIGNMENT 2$ LONGREAL;
HP_CHAR       =   $ALIGNMENT 1$ CHAR;

PROCEDURE HPSWITCHTOCM;
   INTRINSIC;
```

```
PROCEDURE HPSETCCODE;
   INTRINSIC;

PROCEDURE QUIT;
   INTRINSIC;

{End of OUTER BLOCK GLOBAL declarations}

FUNCTION FREAD $ALIAS 'FREAD'$
               (
                       FILENUMBER : HP_SHORTINT;
                ANYVAR TARGET : Hp_GENERIC_BUFFER;
                       TCOUNT : HP_SHORTINT
               ) : HP_SHORTINT
               OPTION UNCHECKABLE_ANYVAR;

VAR
   Hp_proc      : Hp_SCM_PROCEDURE;
   Hp_parms     : Hp_SCM_PARM_DESC_ARRAY;
   Hp_method    : INTEGER;
   Hp_nparms    : INTEGER;
   Hp_funclen   : INTEGER;
   Hp_funcptr   : INTEGER;
   Hp_byte_len_of_parm : Hp_BIT16;
   Hp_cond_code : SHORTINT;
   Hp_status    : Hp_STATUS_TYPE;

VAR  Hp_retval    : SHORTINT;
VAR  Hp_loc_FILENUMBER : HP_SHORTINT;
VAR  Hp_loc_TCOUNT     : HP_SHORTINT;

begin {STUB procedure FREAD}

{*********************************************}
{*                                           *}
{* Generated: WED, OCT 21, 1987, 10:21 AM    *}
{*                                           *}
{* Switch Assist Tool HP30363A.00.00         *}
{*                                             *}
{***********************************************}

{Initialization}

{Set up procedure information--name, lib, etc.}

   Hp_proc.Hp_p_proc_id_type := Hp_Pidt_Name;  {by name}
   Hp_proc.Hp_p_lib          := Hp_System_SL;  {library}
   Hp_proc.Hp_p_proc_name    := 'FREAD           ';

{Set up misc. variables}

   Hp_method := Hp_Method_Normal;  {non-split-stack}
   Hp_nparms := 3;

{Set up length/pointers for functional return if this}
{is a FUNCTION.  Set length to zero, pointer to NIL  }
{if this is not a FUNCTION.                          }

   Hp_funclen := SIZEOF(Hp_retval);
   Hp_funcptr := INTEGER(LOCALANYPTR(ADDR(Hp_retval)));

{Make a local copy of all VALUE parameters}

   Hp_loc_FILENUMBER := FILENUMBER;
   Hp_loc_TCOUNT     := TCOUNT;

{Build the parm descriptor array to describe each}
{parameter.                                      }

{FILENUMBER -- Input Only by VALUE}

   Hp_byte_len_of_parm := 2;

   Hp_parms[0].Hp_pd_parmptr   := ADDR(Hp_loc_FILENUMBER);
   Hp_parms[0].Hp_pd_parmlen   := Hp_byte_len_of_parm;
   Hp_parms[0].Hp_pd_parm_type := Hp_Parm_Value;
   Hp_parms[0].Hp_pd_io_type   := [Hp_input];
```

```
{TARGET -- Output Only by REFERENCE}

   IF TCOUNT < 0 THEN
      Hp_byte_len_of_parm := ABS(TCOUNT)
   ELSE
      Hp_byte_len_of_parm := TCOUNT * 2;

   Hp_parms[1].Hp_pd_parmptr   := ADDR(TARGET);
   Hp_parms[1].Hp_pd_parmlen   := Hp_byte_len_of_parm;
   Hp_parms[1].Hp_pd_parm_type := Hp_Parm_Word_Ref;
   Hp_parms[1].Hp_pd_io_type   := [Hp_output];

{TCOUNT -- Input Only by VALUE}

   Hp_byte_len_of_parm := 2;

   Hp_parms[2].Hp_pd_parmptr   := ADDR(Hp_loc_TCOUNT);
   Hp_parms[2].Hp_pd_parmlen   := Hp_byte_len_of_parm;
   Hp_parms[2].Hp_pd_parm_type := Hp_Parm_Value;
   Hp_parms[2].Hp_pd_io_type   := [Hp_input];

{Do the actual SWITCH call}

HPSWITCHTOCM (Hp_proc,          {Procedure info}
              Hp_method,        {Switch copy method}
              Hp_nparms,        {Number of parameters}
              Hp_parms,         {Parm descriptor array}
              Hp_funclen,       {func ret value length}
              Hp_funcptr,       {addr of func return}
              Hp_cond_code,     {cond code return}
              Hp_status);       {SWITCH status code}

if (Hp_status.Hp_all <> Hp_all_ok) then
   begin {SWITCH subsystem error}
      QUIT (Hp_status.Hp_info);  {handles error codes}
                                 {returned by Switch}
   end; {SWITCH subsystem error}
```

```
HPSETCCODE (Hp_cond_code);

FREAD := Hp_retval;

end; {STUB procedure FREAD}

BEGIN {Program Outer block code}

END. {Program Outer block code}
```

SWAT Quick Reference Summary

The following steps summarize the process of using SWAT to generate Switch stubs automatically and how to use SWAT-generated stubs:

1. To invoke SWAT, type: `RUN SWAT.PUB.SYS`. SWAT uses VPLUS screens to prompt you for the information it needs.
2. Follow these guidelines when filling in a menu:
   a. Use the Cursor Control keys or the Tab key to move from field to field and item to item within a menu.
   b. To complete some fields, you must enter a name or numerical value.
   c. If the field consists of a list of options, place a nonblank character in the space provided next to your choice.
   d. To record information on the menu, press the (Enter) key.
   e. If you have made errors while filling the menu, SWAT will help you resolve them by highlighting the fields containing errors and displaying diagnostic messages.
3. Fill in SWAT's MAIN menu with the following information:
   - Source code file name
   - Target CM procedure name
   - Target procedure parameter names

4. Fill in SWAT's PROCINFO menu with the following information:
   - Location of the target procedure
   - Function return type of the target procedure
   - Privilege level of the target procedure
   - Whether the target procedure returns a condition code
5. For each parameter, fill in SWAT's PARMINFO menu with the following information:
   - Parameter's addressing method
   - Parameter's data type
   - Parameter's I/O type
   - Whether a parameter is an array
6. If a parameter is an array, fill in SWAT's ARRAYLEN menu with the following information:
   - Length value of the array parameter
   - How the length value is to be interpreted
7. If you want to review or edit any of these menu screens, use the F5 and F6 function keys to move backward and forward through the screens.
8. When finished with your review/edit, proceed to the COMMIT screen and press the F2 function key to begin generating the Switch stub source code.
9. When code generation begins, the PROGRESS MESSAGES screen appears and displays any progress, error, and/or result messages associated with the code generation process.
10. When SWAT has generated the source for your Switch stub, it redisplays the MAIN menu screen. You can either press the F8 function key to exit SWAT or return to step 2 to generate another stub.
11. Use the `PASXL` command to compile the SWAT-generated Switch stub. For more information on `PASXL`, refer to the *MPE XL Commands Reference Manual* (32650-90003).

12. Use the Link Editor to link the resulting object code into an NM program file or install it in an NM Executable Library. For more information on the Link Editor, refer to the *Link Editor/XL Reference Manual* (32650-90030).

Special Cases

There are certain situations that the Switch Assist Tool does not handle. Other situations require special consideration before you decide to use SWAT-generated Switch stubs.

Note The following information applies only to switches in the NM—> CM direction. The Switch Assist Tool does not handle mixed-mode procedure calls in the CM—> NM direction. Refer to Chapter 5 for a discussion of how to code CM—> NM stubs manually.

Unsupported Cases

The Switch Assist Tool cannot handle the following:

- Value parameters that are larger than a single element (for example, arrays passed by value) are not handled.
- No pointers of any kind are allowed.
- Plabels, and program labels in general, cannot be passed as parameters.
- CM procedures that execute in split-stack mode cannot be accessed by SWAT-generated stubs.

In Pascal, any size structure can be passed by value to another procedure because a copy is placed on the top of the stack. SPL, on the other hand, does not allow arrays to be passed by value. Because most CM target procedures are written in SPL, the Switch Assist Tool does not allow multi-element parameters, such as arrays, to be passed by value.

For a pointer to be valid, all its possible targets would also have to be available to the called procedure. This is not possible since all available storage within the NM process stack would have to be copied. There may be some cases where an application dictates that a pointer always points to a location within one of the other passed parameters, but you must handle those instances manually.

Passing plabels across modes implies that you can transfer control across modes without using Switch. Since that is not the case, you cannot pass plabels as parameters.

Caution Programming a correct and reliable split-stack Switch stub requires a very high level of knowledge about programming in both modes on MPE XL. It also requires a strict coding discipline in all of the procedures that the target procedure calls, as well as special programming and debugging techniques for error handling and recovery. In addition, all of the warnings and restrictions about Privileged Mode programming apply to split-stack stubs and their target procedures. Therefore, Hewlett-Packard recommends that, if your application needs access to split-stack target procedures, you should request assistance from trained Hewlett-Packard support staff.

Special Case Considerations

There are some conditions that either are undetectable by the Switch Assist Tool or make use of the SWAT-generated source code inadvisable. These situations require modification of SWAT-generated Switch stubs:

- Use of unconverted variables of type real or long
- Overlapping reference parameters
- Specification of reference parameter data lengths at run time
- Option-variable procedures

Unconverted Real or Long Variables

Compatibility Mode has HP 3000 mode floating-point format, while Native Mode supports both the HP 3000 mode format and IEEE floating-point format. You can use the Switch Assist Tool to generate Switch stubs for procedures that have variables of type real or long. If you have not already coded their conversion, then, to guarantee correct results, you must insert calls to the HPFPCONVERT intrinsic into the SWAT-generated stub. You may have to perform the conversion twice, once in preparation for the Switch and again upon returning from it. No conversion is necessary if the calling procedure has already done the conversion or if the caller used HP 3000 Mode floating-point format. Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for information about HPFPCONVERT.

Overlapping Reference Parameters

If reference parameters overlap, unpredictable results can occur. In Native Mode, the parameters will overlap; in Compatibility Mode, however, the parameters are placed one after the other on the CM stack. The Switch Assist Tool cannot detect this situation. If you design calls to CM procedures via Switch, it is your responsibility to deal with the consequences of overlapping reference parameters.

Run-time Specification of Reference Parameter Data Lengths

Reference parameter data lengths must be known at run time in the stub procedure. It is not simply a matter of specifying the maximum length because the length specified is copied to the CM stack and back again to the NM data space. In many instances, it would not be practical, or even possible, to use the maximum length. Such is the case with the *target* parameter (buffer) for FREAD or FWRITE. For many situations, however, it is possible to specify a maximum length. For example, the maximum length of the *formalfiledesignator* parameter in an FOPEN call is never larger than 36 bytes.

If you use a maximum length, you should declare the parameter to Switch as an input/output parameter, even if it is output only. If you declare the parameter as output only, then part of your NM data area can be altered when the CM data area of the declared length is copied back to the NM data area. On the other hand, if you declare the parameter to be input/output, then any portion of the data area that was not changed during execution of the CM target is copied back unchanged to the NM data area.

Several options are available to alleviate the reference parameter data length problem. A discussion of two such options follows.

As one option, you can scan the buffer for a termination character to determine the length. This method is applicable in many cases where the data is terminated by a carriage return (for example, the `COMMAND` intrinsic) or a binary zero (for example, the `GENMESSAGE` intrinsic).

Or, as another option (using SWAT), you can take the length of the buffer from another parameter. In the case of `FREAD/FWRITE`, you specify the length in the call. On the PARMINFO screen, you supply the name of the parameter and then select the `USE NEGATIVE = BYTES RULE` option. SWAT must do some additional processing to use the length value since the length passed in the call is negative if specified in bytes or positive if specified in words. The Switch call requires the length to be specified in positive bytes. So, SWAT performs one of two conversions:

- Negative values (specifying byte lengths) are made positive.
- Positive values (specifying word lengths) are multiplied by two to convert the word count to a byte count.

Option-variable Procedures

The problems involving calls to option-variable procedures include the following:

- No direct HP Pascal/XL equivalent for the `OPTION VARIABLE` option
- Default value options for non-HP Pascal/XL programs
- Interference of `EXTERNAL` declarations with default value options
- Determining if a value parameter was included in the call

The `OPTION VARIABLE` option, as defined in the HP 3000/SPL environment, is not supported in the HP Pascal/XL environment. Although the replacement capability of HP Pascal/XL allows the specification of default values for parameters left out of a procedure call, this capability does not provide the functional equivalent of SPL option variable calls.

For the non-HP Pascal/XL program, the only way for the default values options to function is through the use of an intrinsic file. The Switch Assist Tool does not write the code necessary to place the declaration of each option-variable stub into an intrinsic file.

Even if you manually place the stub's declaration in an intrinsic file, the default values option does not function if there is an `EXTERNAL` declaration for the stub procedure embedded within the calling program's code.

Assuming there is no such `EXTERNAL` declaration and the default value option works correctly, you can give reference parameters a default value of `NIL`. Since `NIL` is never a valid value for reference parameters, this provides an easy method for determining whether the parameter was passed. However, parameters passed by value can have any value since, unlike reference parameters, they do not have to be a valid address. This poses a major problem in distinguishing value parameters included in the call. A default value that will *never* be valid must somehow be assigned in order to identify value parameters that were given default values versus value parameters whose values were specified in the call.

User Options

Pointers are not allowed, and the Switch Assist Tool does not generate them. However, if a pointer always points to an address within another variable that is also being passed in the Switch call, then you can manually code the statements needed to transfer the pointer.

3

NM-to-CM Procedure Calls

This section presents the following topics:

- Review of the flow of control in NM—> CM switches
- Details of NM—> CM switches, including their inner workings, syntax, parameters, and examples
- Special considerations and restrictions that apply to mixed-mode procedure calls
- Testing and debugging considerations

This chapter provides the detailed reference information you will need if you write your own NM-to-CM Switch stubs. Refer to Chapter 5 for further information on writing Switch stubs.

Overview

This overview presents mixed-mode procedure calls in the NM—> CM direction in two ways:

- Schematic flow-of-control diagram
- Stepwise sequence of events

Flow of Control: NM—> CM

The flow of control for mixed-mode procedure calls in the direction NM—> CM is illustrated in Figure 3-1:

Figure F03-01 here.

Figure 3-1. NM—> CM Switch Summary 1

Stepwise Switch to CM

A mixed-mode procedure call in the direction NM—> CM involves the following steps, as indicated in Figure 3-1:

1. The NM code needing to access a CM routine calls Switch, either by means of in-line code or a Switch stub.
2. The in-line code or NM Switch stub sets up the data structures and parameters required by Switch and makes a call to the `HPSWITCHTOCM` intrinsic (call by name), or the `HPLOADCMPROCEDURE` and `HPSWITCHTOCM` intrinsics (call by plabel).
3. `HPSWITCHTOCM` calls the CM routine and passes the parameters, translating addresses and/or copying data to Compatibility Mode as required by their type, size, and location.
4. The CM routine executes using the passed parameters and returns its data values to `HPSWITCHTOCM`.
5. `HPSWITCHTOCM` receives the data values returned by the CM routine, back-translating addresses and recopying data as needed.
6. `HPSWITCHTOCM` returns the translated values to the caller. The stub or in-line code checks whether the Switch operation was successful and then control returns to the NM routine.

Note `HPSWITCHTOCM` is the system intrinsic that makes NM parameters addressable to Compatibility Mode, changes the execution mode, and invokes the CM routine. `HPSWITCHTOCM` and `HPLOADCMPROCEDURE` reside in the NM system library, `NL.PUB.SYS`.

Switch to CM Details

Switch requires the following information to call a CM procedure from an NM procedure:

- Name of the CM SL routine and the library containing it or the 16-bit CM plabel of a currently loaded copy of the routine
- Number of parameters in the calling sequence of the CM SL routine
- Length of each actual parameter
- For each parameter, a designation as an input parameter, an output parameter, or both
- For each parameter, an indication whether the CM SL routine expects that parameter to be a value parameter or a reference parameter; reference parameters require the further specification of a word or byte address
- Whether the CM procedure is a procedure or a function

Note that there are two ways to specify the target procedure, by name or by plabel.

You can supply the ASCII name of the target each time it is called. In this case, Switch automatically loads the procedure from the library the first time it is invoked. Thereafter, Switch uses an internal hash table to find the already loaded procedure.

Instead of specifying the target by name and encountering the lookup overhead, you can call an intrinsic procedure (`HPLOADCMPROCEDURE` or `LOADPROC`) to obtain the target's plabel and supply that to Switch instead of the name. You must save the plabel for use in subsequent calls.

Switch needs the parameter length information for parameters because, in most cases, it must copy data from the NM data space to the CM stack. If the target procedure is callable in split-stack mode (requiring Privileged Mode), and if the reference parameters are close enough to each other to fit in a CM extra data segment, and if the reference parameters are big enough to justify the overhead of a copy operation, then you can request that `HPSWITCHTOCM` wrap the reference parameters in an extra data segment and call the CM procedure in split-stack mode.

HPSWITCH TOCM Intrinsic

The mechanism that enables mixed-mode calls in the NM—> CM direction is the `HPSWITCHTOCM` intrinsic.

Specifically, `HPSWITCHTOCM` does the following:

- Copies value parameters to the CM stack
- Translates NM addresses of reference parameters to CM addresses when they are within the CM data segment (split-stack mode only)
- Copies NM reference parameters to the CM data segment if they are input parameters, converts NM addresses to CM addresses, then copies the parameters back to the NM data space
- Changes the execution mode
- Calls the CM procedure specified by the NM caller

There are three methods of passing reference parameters to the target procedure:

- Normal method
- Split-stack method
- No-copy method

The normal method is the only one available to callers with execution level three (standard user level). This method does the following:

- Allocates room for the NM parameter(s) in the CM stack
- Copies the contents of the NM parameter(s) to the CM stack (if input parameters) and allocates space for output parameters
- Copies the CM results back to the NM parameter(s) upon target completion (if output parameters and if not already in the CM stack)

Caution `HPSWITCHTOCM` does no checking for overlapping reference parameters. On MPE V/E-based systems, you may have used overlapping reference parameters to obtain side effects since both copies could share the overlap area. In Native Mode, the parameters will overlap. In Compatibility Mode, however, the parameters will be placed one on top of the other on the CM stack. If you rely on this particular effect, the results will not be as expected.

If a procedure is split-stack callable, `HPSWITCHTOCM` encapsulates the reference parameters within an extra data segment and calls the target procedure with DB at the base of the extra data segment. When the CM target procedure returns, the extra data segment is released and the DB register is set back to its previous value.

No extra data segment is created, even if requested, if any of the following conditions applies:

- There are no reference parameters (DB is unchanged).
- All reference parameters are already in the CM stack (DB points to the CM stack).
- The reference parameters are either too large or too distant from one another to be encapsulated by an extra data segment.
- The total size of the reference parameters is less than a fixed threshold (300 bytes), in which case copying them to the CM stack is quicker.
- An extra data segment cannot be obtained, in which case the parameters are copied to the CM stack.

The no-copy method is the same as the split-stack method except that the threshold check is omitted.

Note Both the split-stack and the no-copy methods require Privileged Mode.

The syntax of the HPSWITCHTOCM intrinsic and detailed explanations of its parameters are given in the following paragraphs. Also provided are examples of switches in the NM—> CM direction.

Syntax

Prior to calling the HPSWITCHTOCM intrinsic, your programming language may require you to declare it. In HP Pascal/XL, the declaration is as follows:

```
PROCEDURE HPSWITCHTOCM; INTRINSIC;
```

Next comes an example of an HP Pascal/XL call to HPSWITCHTOCM:

```
HPSWITCHTOCM(proc, method, nparms, parms, fretlen,
             fretval, condcode, status);
```

You call the HPSWITCHTOCM intrinsic with eight parameters. These parameters provide Switch with the following information:

- Name and CM library or the plabel of the target procedure.
- Whether the target procedure runs in normal, split-stack, or no-copy mode.
- Number of parameters being passed to the target procedure.
- Array of records, each record containing a description of one parameter being passed, including:
  - Pointer to the parameter; that is, a reference to where the parameter begins in NM memory (value parameters larger than one byte must be 16-bit aligned).
  - Length (size) of the parameter in bytes (must be positive, nonzero integer <= 2 ** 16).
  - For reference parameters, the stub must specify either a byte or word address and also indicate whether the parameter is an input and/or output parameter.
- Length of the function return value (0 if not a function).
- Pointer to the function return value (nil if not a function).
- CM condition code value to be returned from the target procedure.
- Status record to report on HPSWITCHTOCM's operation.

Note

If the SPL procedure is option variable (see the MPE V/E *Intrinsics Reference Manual* (32033-90007) or the header of a non-intrinsic SPL procedure), you must construct the option variable mask and pass it as the last parameter. See Example 3-11 for an illustration.

All this information is key to the correct interpretation of data, return values, and status information when you make mixed-mode procedure calls.

Parameters

A detailed explanation of the parameters of the `HPSWITCHTOCM` intrinsic follows.

Required parameters are shown in **boldface**; optional parameters are shown in *italics*.

proc **record (required)**

Passes the target CM procedure identifier, which you can specify either by a library to search and an ASCII name of up to 16 characters or by a CM plabel (obtained from the `HPLOADCMPROCEDURE` or `LOADPROC` intrinsic).

The structure of the **proc** record varies, depending on how the target procedure is identified. All variants have a **p_proc_id_type** field. The HP Pascal/XL declaration of this record follows:

```
TYPE
   BIT8     = 0..255;
   BIT16    = 0..65535;
   BIT8_A1  = $ALIGNMENT 1$ BIT8;
   BIT16_A1 = $ALIGNMENT 1$ BIT16;
   CM_PROC_NAME = PACKED ARRAY[1..16] OF CHAR;

scm_procedure = packed record

{pidt_known, pidt_load, pidt_plabel}
    case p_proc_id_type  :  bit8 of

{proc found by number}
       pidt_known : ( p_fill : bit8_a1;
                      p_proc_id : bit16_a1 );

{proc found by name}
       pidt_load : (
   {system, pub, or group library}
                      p_lib : bit8_a1;
   {ASCII name left justified & blank padded}
                      p_proc_name : cm_proc_name);
```

```
{proc found by plabel}
       pidt_plabel : (
   {proc's CM plabel}
                       p_plabel : bit16_a1);

    end;
```

If you designate the target CM procedure by means of an ASCII name, then the following apply:

- The value of the **p_proc_id_type** field is 1.
- The search library is designated in a **p_lib** field whose value can be 0 for the system SL, 1 for the logon pub SL, 2 for the logon group SL, 3 for the program's pub SL, or 4 for the program's group SL.
- The ASCII name is designated in the **p_proc_name** field and can be up to 15 bytes long. The name is padded on the right with blanks to a length of 16 bytes.
- The storage layout of the **proc** record is as follows:

```
Word +-------------+----------+-------------------+
   0 | p_proc_id_  | p_lib    | p_proc_name       |
     | type (8 bits)| (8 bits) |  (16 bytes)       |
     +-------------+----------+-------------------+
   1 | p_proc_name (cont.)                        |
     |                                            |
     +--------------------------------------------+
   2 | p_proc_name (cont.)                        |
     |                                            |
     +--------------------------------------------+
   3 | p_proc_name (cont.)                        |
     |                                            |
     +------------------------+-------------------+
   4 | p_proc_name (cont.)    | unused            |
     |                        |                   |
     +------------------------+-------------------+
```

The minimum alignment for this layout is an 8-bit boundary.

Note Switches by name involve high system overhead on the first call per name, but substantially lower overhead on each subsequent call for that name. The `HPSWTONMNAME`, `HPSWITCHTOCM`, `HPLOADCMPROCEDURE`, and `HPLOADNMPROC` intrinsics perform a hashing function on the name of the other-mode procedure and store the plabel for that procedure in a system internal hash table. The `LOADPROC` intrinsic, on the other hand, does not perform any hashing and, consequently, involves high system overhead every time it is called.

If you designate the target CM procedure by means of a plabel, then the following apply:

- The value of the **p_proc_id_type** field is 2.
- A **p_plabel** field designates the plabel.
- The **proc** record has the following storage layout:

```
Word +--------------+----------------+--------------+
   0 | p_proc_id_   | p_plabel       | unused       |
     | type (8 bits)| (16 bits)      |              |
     +--------------+----------------+--------------+
   1 | unused                                       |
     |                                              |
     +----------------------------------------------+
   2 | unused                                       |
     |                                              |
     +----------------------------------------------+
   3 | unused                                       |
     |                                              |
     +----------------------------------------------+
   4 | unused                                       |
     |                                              |
     +----------------------------------------------+
```

The minimum alignment for this layout is an 8-bit boundary.

method **32-bit signed integer by value (required)**

Value indicating whether the CM target procedure runs in normal, split-stack, or no copy mode. The valid values and their meanings follow:

0	Normal (non-split-stack) Method
1	Split-stack Method
2	No-copy Method

Note The split-stack method works as follows:

If all parameters are within the CM stack, or the length of the reference parameters is less than the threshold, then use normal method. Otherwise, wrap the reference parameters in an extra data segment if the reference parameters are outside of the CM stack (split-stack method). The no-copy method is the same as the split-stack method, except that the threshold check is omitted. Both the split-stack and the no-copy method require Privileged Mode.

nparms **32-bit signed integer by value (required)**

The number of parameters you are passing to the CM target procedure.

parms **array of records (required)**

Passes descriptions of each parameter being passed to the CM target procedure. Each parameter is located and described by a record in this array. The fields of the record are as follows:

pd_parmptr **64-bit address**

The NM virtual address of the beginning of the parameter to be passed.

pd_parmlen **16-bit unsigned integer**

The length in bytes of the item that **pd_parmptr** points to. Together with that pointer, **pd_parmlen** specifies the data area being described. The following restriction applies:

1 <= **pd_parmlen** <= (2**16)

pd_parm_type **16-bit unsigned integer**

Provides information needed to determine the kind of mapping to be performed during the transfer of the parameter. The allowed mappings are as follows:

0 Value parameter

1 Reference parameter requiring word address

2 Reference parameter requiring byte address

pd_io_type **32-bit signed integer**

Applies only to reference parameters and specifies whether such a parameter is input, output, or both. Switch uses this information to minimize the amount of data transfer when data must be copied.

Only the two high-order bits in this word are set. Set bit (0:1)=1 to designate an input parameter. Set bit (1:1)=1 to designate an output parameter. Good programming practice dictates that you designate a parameter as being input, output, or both.

The minimum alignment of the **parms** record is 32-bit word boundary, and the storage layout of this record is as follows:

```
Word +-----------------------------------------------+
   0 | pd_parmptr                                    |
     | (8 bytes)                                     |
     +                                               +
   1 |                                               |
     |                                               |
     +---------------------+-------------------------+
   2 | pd_parmlen          | pd_parm_type            |
     | (16 bits)           | (16 bits)               |
     +---------------------+-------------------------+
   3 | pd_io_type                                    |
     | (32 bits)                                     |
     +-----------------------------------------------+
```

Note The **pd_parmptr** field is a 64-bit address that is currently supported only in HP Pascal/XL.

Table 3-1 summarizes, according to MPE XL-supported language, the data alignments of the common data types.

Table 3-1. Data Alignment By XL Language

Data Type	HP Pascal/XL	COBOL II/XL	FORTRAN 77/XL
16-bit signed integer (I16)	halfword	halfword	halfword
32-bit signed integer (I32)	word	word	word
64-bit signed integer (I64)	doubleword	doubleword	doubleword
16-bit unsigned integer (U16)	halfword	halfword	halfword
32-bit unsigned integer (U32)	word	word	word
64-bit unsigned integer (U64)	doubleword	doubleword	N/A
32-bit real (R32)	word	N/A	word
64-bit real (R64)	doubleword	N/A	doubleword
Boolean (B)	byte	byte	byte
Character (C)	byte	byte	byte
32-bit address (@32)	word	word	word
64-bit address (@64)	doubleword	doubleword	N/A
Array (A)	Determined by element type	Determined by element type	Determined by element type
Record (REC)	Boundary of most restrictive element	word	word

fretlen *32-bit signed integer by value (optional)*

The length in bytes of the optional functional return value.

The default is 0.

fretval *record (optional)*

A single-element record containing a 32-bit NM pointer to the beginning of the area to which the optional functional return value is returned.

The HP Pascal/XL declaration of this record is as follows:

```
LOCALANYPTRREC = RECORD
   (rtn_addr: LOCALANYPTR);
END;
```

The default value of this pointer is nil. If *fretval* is omitted, its place in the parameter list is taken by nil.

condcode *16-bit signed integer by reference (optional)*

Returns the condition code that the target CM procedure returns. Valid values are in the range 0 ... 2.

The default is nil. If *condcode* is omitted, its place in the parameter list is taken by nil (32 bits of zero) and no condition code is returned.

status *status-record (optional)*

Returns the status of the intrinsic call. If no errors or warnings are encountered, *status* returns 32 bits of zero. If errors or warnings are encountered, *status* is interpreted as two 16-bit fields. The HP Pascal/XL declaration of this record is as follows:

```
XLSTATUS = RECORD
    CASE INTEGER OF
      0 : (all : INTEGER);
      1 : (info   : SHORTINT;
           subsys : SHORTINT);
END;  {record}
```

Bits (0:16) comprise *status.info*. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise *status.subsys*. The value represented by these bits defines the subsystem that set the status information. The Switch identification number is 100.

The values of *status.info* that can be returned from a call to `HPSWITCHTOCM` are listed in `***<xref T03-02>: undefined***`.

Table 3-2. HPSWITCHTOCM Status Returns (Page 1)

Status Code	Meaning
-20	Invalid **method** parameter value (not in range 0 . . . 2)
-30	Parameter bounds violation
-40	Invalid number of parameters
-50	Invalid parameter length
-60	Privileged operation error
-80	Invalid procedure type
-90	Non-extant plabel
-110	CM Switch-by-name table exhausted
-120	Procedure not loaded
-130	Invalid status parameter
-150	Invalid **proc** address
-152	Invalid **parms** address
-154	Invalid parameter address
-156	Invalid **pd_parm_type** value
-158	Invalid parameter I/O specification
-160	Invalid function return length
-162	Invalid function return address
-164	Invalid function return specification
-166	Invalid condition code address
-168	Invalid *status* address

Table 3-2. CM Status Returns (Page 2)

Status Code	Meaning
-190	Invalid **proc** length
-194	Invalid *lib* length
-200	Invalid function type
-210	Invalid parameter type
-250	Invalid number of arguments
-260	Invalid process ID
-270	Invalid Switch-to-NM executor
-290	FINDPROC error
-300	Illegal reference parameter alignment (No copy method where the actual parameter is byte-aligned and the **pd_parm_type** = 1, indicating a reference parameter requiring a word address)

Note Since call by name can result in a call to the `LOADPROC` intrinsic, Switch can also return CM Loader errors. See Table 3-3.

Caution Since `HPSWITCHTOCM` can return information on the success of its execution in the *status* parameter, it is good programming practice to specify this parameter and check its value after the intrinsic call. If an error or warning condition is encountered and you did not specify the *status* parameter, `HPSWITCHTOCM` causes the calling process to abort.

`HPSWITCHTOCM` makes the following checks on its parameters:

- If a plabel is used, it is range-checked.
- The number of the target's parameters must be less than or equal to 32 (the maximum number of parameters).
- If the split-stack or no-copy method is used, the caller must be operating at either execution level one or zero.
- The Switch method must be valid (either 0 for normal, 1 for split-stack, or 2 for no-copy).
- For each parameter, the **pd_parm_type** value must be valid (0, 1, or 2), and the **pd_parm_len** value must be positive and <= 2 ** 16.

Switch relies on the hardware to check for the following:

- Invalid data area
- Target procedure not user-callable (execution level violation)

If you do not specify the *status* parameter and the switching operation is successful, the `HPSWITCHTOCM` intrinsic returns to the calling routine. However, if an error occurs, `HPSWITCHTOCM` assumes the caller is HP Pascal/XL and attempts to escape to the caller's RECOVER block. If there is no RECOVER block, the program is aborted. Refer to TRY/RECOVER in the *HP Pascal/XL Reference Manual* (31502-90001).

HPLOADCM PROCEDURE Intrinsic

Switching by name incurs the overhead of forming a hash probe out of the procedure's name and finding the name in a hash table in order to obtain the procedure's plabel.

You can eliminate this overhead by obtaining the target procedure's plabel through the Switch support intrinsic function `HPLOADCMPROCEDURE`. You then supply the result returned by `HPLOADCMPROCEDURE` as the value of the **proc** parameter in the call to the `HPSWITCHTOCM` intrinsic.

Note This method is to your advantage only if you make many calls to the same procedure. Otherwise, no overhead is saved.

Syntax

In HP Pascal/XL, the declaration of the `HPLOADCMPROCEDURE` intrinsic is as follows:

```
FUNCTION HPLOADCMPROCEDURE : SHORTINT; INTRINSIC;
```

Next, an example of an HP Pascal/XL call to this function:

```
plabel := HPLOADCMPROCEDURE(proc, lib, status);
```

You call the `HPLOADCMPROCEDURE` intrinsic with three parameters. These parameters provide the following information:

- ASCII procedure name
- Indicator of the CM SL to be searched
- Status record to report on `HPLOADCMPROCEDURE`'s operation

Parameters

A detailed explanation of the parameters of the `HPLOADCMPROCEDURE` intrinsic follows.

Required parameters are shown in **boldface**; optional parameters are shown in *italics*.

proc **character array (required)**

Passes an ASCII procedure name, left-justified and blank-padded. The name can have a maximum of 15 characters. The name is padded with blanks to a length of 16.

lib *8-bit unsigned integer (optional)*

Passes indicator of the CM SL to be searched. The valid values are as follows:

0	Search the system SL only.
1	Search the logon account SL, then the system SL.
2	Search the logon group SL first, the logon account SL second, and the system SL last.
3	Search the program file's account SL, then the system SL.
4	Search the program file's group SL first, the program file's account SL second, and the system SL last.

The system SL is the default.

status *status-record (optional)*

Returns the status of the intrinsic call. If no errors or warnings are encountered, *status* returns 32 bits of zero. If errors or warnings are encountered, *status* is interpreted as two 16-bit fields. The HP Pascal/XL declaration of this record is as follows:

```
XLSTATUS = RECORD
    CASE INTEGER OF
       0 : (all : INTEGER);
       1 : (info   : SHORTINT;
            subsys : SHORTINT);
END;  {record}
```

Bits (0:16) comprise *status.info*. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16) comprise *status.subsys*. The value represented by these bits defines the subsystem that set the status information. The Switch subsystem identification number is 100.

The values of *status.info* that can be returned from a call to `HPLOADCMPROCEDURE` (or `HPUNLOADCMPROCEDURE`) are listed in Table 3-3. These numbers are derived as follows: - |CM Loader Error Number| - 1000.

Caution Since `HPLOADCMPROCEDURE` can return information on the success of its execution in the *status* parameter, it is good programming practice to specify this parameter and check its value after the intrinsic call. If an error or warning condition is encountered and you did not specify the *status* parameter, `HPLOADCMPROCEDURE` causes the calling process to abort.

Table 3-3. CM Loader Status Returns (Page 1)

Status Code	Meaning
-1020	Illegal library search.
-1021	Unknown entry point.
-1022	The TRACE subsystem is not present.
-1023	The stack size is too small.
-1024	Maxdata is greater than 32K.
-1025	The data segment is greater than the maxdata segment.
-1026	The program is loaded in the opposite mode.
-1027	SL binding error.
-1028	Invalid system SL file.
-1029	Invalid public SL file.
-1030	Invalid group SL file.
-1031	Invalid program file.
-1032	Invalid list file.
-1033	The code segment is greater than the system maximum.
-1034	The program uses more than one extent.
-1035	The data segment is greater than 32K.
-1036	The data segment is greater than the system maximum.
-1037	The number of code segments is greater than 63.
-1038	The number of code segments is greater than the system maximum.
-1039	Illegal capability.
-1040	Too many procedures are loaded.
-1041	Unknown procedure name.
-1042	Invalid procedure number.

Table 3-3. CM Loader Status Returns (Page 2)

Status Code	Meaning
-1043	Illegal procedure unload.
-1044	Illegal SL capability.
-1045	Invalid entry point.
-1050	Unable to open system SL file.
-1051	Unable to open public SL file.
-1052	Unable to open group SL file.
-1053	Unable to open program file.
-1054	Unable to open list file.
-1055	Unable to close system SL file.
-1056	Unable to close public SL file.
-1057	Unable to close group SL file.
-1058	Unable to close program file.
-1059	Unable to close list file.
-1060	EOF or I/O error on system SL file.
-1061	EOR or I/O error on public SL file.
-1062	EOF or I/O error on group SL file.
-1063	EOF or I/O error on program file.
-1064	EOF or I/O error on list file.
-1065	Unable to obtain CST entries.
-1066	Unable to obtain process DST entry.
-1067	Unable to obtain mail data segment.
-1068	Unable to obtain working set.
-1069	Unable to obtain CSTX entries.

Table 3-3. CM Loader Status Returns (Page 3)

Status Code	Meaning
-1070	Segment Table overflow.
-1071	Unable to obtain sufficient DL storage.
-1072	ATTIO error.
-1073	Unable to obtain virtual memory.
-1074	Directory I/O error.
-1075	Print I/O error.
-1076	Illegal DLSIZE.
-1080	The program is already allocated.
-1081	Illegal program allocation.
-1082	The program is not allocated.
-1083	Illegal program deallocation.
-1084	The procedure is already allocated.
-1085	Illegal procedure allocation.
-1086	The procedure is not allocated.
-1087	Illegal procedure deallocation.
-1092	ALLOCATE/DEALLOCATE from non-system disc.
-1093	Unable to mount program's home volume set.
-1094	Unable to mount system SL's home volume set.
-1095	Unable to mount private SL's home volume set.
-1096	Unable to mount group SL's home volume set.
-1097	Unable to load remote program.
-1098	Unable to convert old format.
-1099	Unable to obtain DST for logical map.

Table 3-4. Table 3-3. CM Loader Status Returns (Page 4)

Status Code	Meaning
-1100	There are too many mapped segments.
-1101	The SEGMAP is too big.
-1102	Unable to expand SEGMAP.
-1103	There are too many dynamic loads on the procedure.

HPUNLOADCM PROCEDURE Intrinsic

Procedures whose plabels are obtained via the HPLOADCMPROCEDURE intrinsic function are automatically unloaded when the process terminates. However, you can use the HPUNLOADCMPROCEDURE intrinsic to do the unloading before the process terminates.

Syntax

In HP Pascal/XL, the declaration of the HPUNLOADCMPROCEDURE intrinsic is as follows:

```
PROCEDURE HPUNLOADCMPROCEDURE; INTRINSIC;
```

Next is an example Pascal/XL call to HPUNLOADCMPROCEDURE:

```
HPUNLOADCMPROCEDURE (proc, lib, status);
```

Parameters

The parameters of the HPUNLOADCMPROCEDURE intrinsic are identical to those of the HPLOADCMPROCEDURE intrinsic function.

Note You can use the LOADPROC and UNLOADPROC as functional equivalents of HPLOADCMPROCEDURE and HPUNLOADCMPROCEDURE, respectively. HPLOADCMPROCEDURE performs a hashing function on the name of the other-mode procedure and stores the plabel for that procedure in a system internal hash table. LOADPROC performs no hashing. Consequently, while both intrinsics involve high system overhead on the first call per name, HPLOADCMPROCEDURE involves substantially lower overhead on each subsequent call for that name. Refer to the appropriate entries in the *MPE XL Intrinsics Reference Manual* (32650-90028) for further details.

Examples: NM to CM and Return

Now consider an example of the NM—> CM mixed-mode switching process. Suppose there is an SPL procedure CONVERTDATE that resides in a CM SL and that you want to access from MPE XL code. CONVERTDATE converts an input parameter `FROMDATE` to an output parameter `TODATE`, based on the values of the `FROMFORMAT` and `TOFORMAT` parameters.

The SPL declaration of the procedure is as follows:

```
+---------------------------------------------------------+
|                                                         |
|PROCEDURE CONVERTDATE (FROMDATE, TODATE,                 |
|                      FROMFORMAT, TOFORMAT, DATELENGTH);|
|          VALUE       FROMFORMAT, TOFORMAT, DATELENGTH; |
|          BYTE ARRAY  FROMDATE, TODATE;                 |
|          INTEGER     FROMFORMAT, TOFORMAT, DATELENGTH; |
|                                                         |
+---------------------------------------------------------+
```

The `DATELENGTH` parameter contains the number of bytes in the date.

The `FROMFORMAT`, `TOFORMAT`, and `DATELENGTH` parameters are declared to be value parameters, while `FROMDATE` and `TODATE` are passed by reference.

Because the CONVERTDATE procedure resides in a CM SL, you can write a Switch stub that calls the `HPSWITCHTOCM` intrinsic, which, in turn, calls CONVERTDATE.

Figure 3-2 illustrates the purpose of the CONVERTDATE Switch stub.

Figure F03-02 here.

Figure 3-2. HPSWITCHTOCM Example, CONVERTDATE

Recall that Switch stubs make the mixed-mode calling process transparent to the calling program so that program need not change. This is possible because the stub procedure name is identical to the CM target procedure name, and the stub procedure parameters and their types are also identical to those of the target procedure. This allows the recompiled application to continue to call CONVERTDATE without any changes to the source code of the application.

The following sample procedure declarations demonstrate these aspects of achieving mixed-mode calling transparency. The first extract is the declaration portion of an SPL procedure in the CM SL:

```
  +---------+
+-| MPE V/E |-------------------------------------------+
| +---------+                                            |
|                                                        |
|PROCEDURE CONVERTDATE (FROMDATE, TODATE,                |
|                       FROMFORMAT, TOFORMAT, DATELENGTH);|
|          VALUE        FROMFORMAT, TOFORMAT, DATELENGTH; |
|          BYTE ARRAY   FROMDATE, TODATE;                 |
|          INTEGER      FROMFORMAT, TOFORMAT, DATELENGTH; |
|                                                        |
|                                                        |
+-------------------------------------------------------+
```

The second extract is the HP Pascal/XL declaration portion of a Switch stub that enables you to call the corresponding CM SL procedure:

```
  +--------+
+-| MPE XL |--------------------------------------------+
| +--------+                                             |
|                                                        |
|TYPE                                                    |
|   DATE_BUFFER : PACKED ARRAY [1..80] OF CHAR;          |
|                                                        |
|                                                        |
|PROCEDURE CONVERTDATE(VAR FROMDATE, TODATE : DATE_BUFFER;|
|            FROMFORMAT, TOFORMAT, DATELENGTH : SHORTINT);|
|                                                        |
|                                                        |
|                                                        |
+-------------------------------------------------------+
```

Note the ways in which the MPE XL declaration makes the stub and the switching process transparent to any program that calls the SPL procedure:

- The Switch stub procedure name is identical to that of the SPL procedure.
- The types of the Switch stub parameters are declared to correspond to those of the SPL procedure:
  - Value parameters correspond to value parameters; likewise, reference parameters correspond.

- □ The data types of the corresponding parameters are compatible.

Another responsibility of the Switch stub is to set up the parameters required by the appropriate Switch intrinsic. In this instance, that is the `HPSWITCHTOCM` intrinsic. Here, again, is a sample call to `HPSWITCHTOCM`:

```
HPSWITCHTOCM(proc, method, numparms, parms, funcreturnlen,
             funcvalue, conditioncode, userstatus)
```

The parameters that the CONVERTDATE Switch stub must set up before it can call the `HPSWITCHTOCM` intrinsic convey to Switch the following information:

- Name and CM library or plabel of the target procedure.
- Whether the target procedure runs in normal, split-stack, or no-copy mode.
- Number of parameters being passed to the target procedure.
- Array of records, each containing a description of one parameter being passed, including:
  - □ Pointer to the parameter; that is, a reference to where the parameter begins in NM memory. Value parameters larger than one byte must be 16-bit aligned.
  - □ Length (size) of the parameter in bytes (must be positive integer <= 2 ** 16).
  - □ For reference parameters, the stub must specify either a byte or word address and also indicate whether the parameter is an input and/or output parameter.
  - □ Length of the function return value (0 if not a function).
  - □ Pointer to the function return value (nil if not a function).
  - □ CM condition code value to be returned from the target procedure.
  - □ Status record to report on `HPSWITCHTOCM`'s operation.

In the course of Example 3-1A, the necessary constants, types, and variables are declared. The body of the stub follows in Example 3-1B. Examples 3-1A and 3-1B together constitute the CONVERTDATE stub.

Example 3-1A. Declarations Portion of CONVERTDATE Stub

```
$standard_level 'EXT_MODCAL'$
$subprogram$
$os 'MPE/XL'$
Program XAMPL31A(input, output);

const   { * * * PROC parameter * * * }

{The OS finds procedures by number, by name, or by plabel}

    pidt_known = 0;  {it is found by number           }
    pidt_load  = 1;  {it is found by name             }
    pidt_plabel = 2; {it is specified by its CM plabel}

{Which library is the procedure in?}

    system_sl        =  0;
    logon_pub_sl     =  1;
    logon_group_sl   =  2;
    pub_sl           =  3;
    group_sl         =  4;

{max CM procedure name length}

    length_cm_proc_name = 16;

type

    bit8                  = 0..255;
    bit16                 = 0..65535;
    bit8_a1               = $ALIGNMENT 1$ bit8;
    bit16_a1              = $ALIGNMENT 1$ bit16;

{type declaration for procedure names}

    cm_proc_name = packed array [1..length_cm_proc_name]
                   of char;
```

```
{defining type of HPSWITCHTOCM proc parameter}

    scm_procedure = packed record
        case p_proc_id_type  :  bit8 of
   {proc found by number}
            pidt_known : ( p_fill : bit8_a1;
                           p_proc_id : bit16_a1 );
   {proc found by name}
            pidt_load : (
      {system, pub, or group library}
                          p_lib : bit8_a1;
      {ASCII name left justified & blank padded}
                          p_proc_name : cm_proc_name);
   {proc found by proc's CM plabel}
            pidt_plabel : ( p_plabel : bit16_a1);
         end;
```

Example 3-1A. Declarations Portion of CONVERTDATE Stub, continued

```
const  { * * * METHOD parameter * * * }

{defining HPSWITCHTOCM method parameter}

    method_normal = 0; {non-split-stack}
    method_split  = 1; {if all parameters within cm stack
                           or ref parm length < threshold
                        then use method_normal
                        else wrap ref parms in extra data
                             segment                     }
    method_no_copy = 2; {use split stack if ref parms
                         outside of cm stack             }

const  { * * * NUM_PARMS parameter * * * }

    max_target_parms = 32;  {max # of target parameters}

const  { * * * PARMS parameter * * * }

{What is the parameter type?}
```

```
    parm_type_value     =   0;  {value parameter         }
    parm_type_word_ref  =   1;  {word address is required}
    parm_type_byte_ref  =   2;  {byte address is required}

type

{defining type of indicator as byte or word address}

    scm_parm_type   =   bit16;

{defining type of indicator as input and/or output      }
{parameter                                              }

    scm_io_type   =   set of ( INPUT_PARM, OUTPUT_PARM );

{defining individual record of HPSWITCHTOCM parms }
{parameter; parms is array describing the stub    }
{parameters; each record describes a parameter    }

    parm_desc   =   packed record

       pd_parmptr : globalanyptr;  {where parameter found}
       pd_parmlen : bit16;                 {size in bytes}
       pd_parm_type : scm_parm_type;
       {value parm or byte or word address reference parm}
       pd_io_type :   scm_io_type;   {input and/or output}

     end;

{defining type of HPSWITCHTOCM parms parameter}

    scm_parm_desc_array = array [1..max_target_parms]
                          of parm_desc;
```

Example 3-1A. Declarations Portion of CONVERTDATE Stub, continued

```
const   { * * * CONDITION_CODE parameter * * * }
```

```
    ccg  =  0;
    ccl  =  1;
    cce  =  2;

type

    ccode_type = shortint;

const  { * * * STATUS constant * * * }

   all_ok = 0;  {used in status check}

type   { * * * STATUS parameter * * * }

    xlstatus = record
       case integer of
          0 : (all : integer);
          1 : (info : shortint;
               subsys : shortint);
       end;

type  { * * * Switch stub parameters * * * }

   date_buffer = packed array[1..80] of char;

{declaring intrinsic procedures -- externals}

procedure HPSWITCHTOCM; intrinsic;

procedure HPSETCCODE; intrinsic;

procedure QUIT; intrinsic;

{declaring Switch stub header}

procedure CONVERTDATE (var FROMDATE, TODATE : date_buffer;
              FROMFORMAT, TOFORMAT, DATELENGTH : shortint);
```

```
var

    proc   :     scm_procedure;         {target proc name}
    parms  :     scm_parm_desc_array;
                     {describes target proc's parameters}
    method :     integer;                   {method of call}
    nparms :     integer;       {# of target's parameters}
```

Example 3-1A. Declarations Portion of CONVERTDATE Stub, continued

```
{declaring return parameters}

    funclen :   integer;          {length of return value}
    funcptr :   integer;         {pointer to return value}
    cond_code : ccode_type;  {how condition code returned}
    status :    xlstatus;    {how MPE XL returns warnings}

{declaring local variables}

    loc_fromformat :     bit16;             { FROMFORMAT }
    loc_toformat   :     bit16;               { TOFORMAT }
    loc_datelength :     bit16;             { DATELENGTH }

{end example 3-1A, completing the declaration portion}
```

Example 3-1B. Body of CONVERTDATE Stub

```
begin

{format and length parameters are copied into local
 variables; local copies used since first 4 parms of proc
 are put in registers; Switch wants address but value in
 register won't have address; to get address, you must
 make local copy}

   loc_fromformat  :=  FROMFORMAT;
   loc_toformat    :=  TOFORMAT;
   loc_datelength  :=  DATELENGTH;

{the Switch variables are initialized}
```

```
   proc.p_proc_id_type := pidt_load;   {find proc by name}
   proc.p_lib       := pub_sl;    {look in PUB SL (LIB=P)}
   proc.p_proc_name := 'CONVERTDATE ';    {procedure name}
   method    := method_normal;  {NOT split-stack callable}
   nparms    := 5;                  {number of parameters}
                   {unless option variable, when option}
                   {variable, add 1 to nparms          }
   funclen   := 0;       {not a function--no return value}
   funcptr   := 0;       {not a function--no return value}

{In the next sequence, "describe" involves the following:}
{   1) give a pointer to the parameter's location   }
{   2) give the length (size) of the parameter      }
{   3) indicate whether value or reference parameter}
{      IF a reference parameter, THEN               }
{   4) indicate whether input and/or output parameter}


{describe FROMDATE -- input by reference}

{determine pointer to parameter's location; addr        }
{function takes parameter as argument and returns address}

   parms[1].pd_parmptr   := addr(FROMDATE);

{determine length of parameter;                        }
{sizeof function takes parameter as argument and returns}
{number of bytes in parameter                          }

   parms[1].pd_parmlen   := sizeof(FROMDATE);

{reference parameter requiring byte address}

   parms[1].pd_parm_type := parm_type_byte_ref;

{input parameter}
```

```
      parms[1].pd_io_type   := [input_parm];
```

Example 3-1B. Body of CONVERTDATE Stub, continued

```
   {describe TODATE -- output by reference}

   {determine pointer to parameter's location; addr        }
   {function takes parameter as argument and returns address}

      parms[2].pd_parmptr   := addr(TODATE);

   {determine length of parameter;                         }
   {sizeof function takes parameter as argument and returns}
   {number of bytes in parameter                           }

      parms[2].pd_parmlen   := sizeof(TODATE);

   {reference parameter requiring byte address}

      parms[2].pd_parm_type := parm_type_byte_ref;

   {output parameter}

      parms[2].pd_io_type   := [output_parm];


   {describe TOFORMAT -- input by value}

   {determine pointer to parameter's location; addr        }
   {function takes parameter as argument and returns address}

      parms[3].pd_parmptr   := addr(loc_toformat);

   {determine length of parameter;                         }
   {sizeof function takes parameter as argument and returns}
   {number of bytes in parameter                           }

      parms[3].pd_parmlen   := sizeof(TOFORMAT);
```

```
{value parameter}

   parms[3].pd_parm_type := parm_type_value;

{input parameter}

   parms[3].pd_io_type := [input_parm];


{describe FROMFORMAT -- input by value}

{determine pointer to parameter's location; addr         }
{function takes parameter as argument and returns address}

   parms[4].pd_parmptr   := addr(loc_fromformat);

{determine length of parameter;                         }
{sizeof function takes parameter as argument and returns}
{number of bytes in parameter                           }
```

Example 3-1B. Body of CONVERTDATE Stub, continued

```
   parms[4].pd_parmlen   := sizeof(FROMFORMAT);

{value parameter}

   parms[4].pd_parm_type := parm_type_value;

{input parameter}

   parms[4].pd_io_type := [input_parm];

{describe DATELENGTH -- input by value}

{determine pointer to parameter's location; addr         }
{function takes parameter as argument and returns address}

   parms[5].pd_parmptr   := addr(loc_datelength);
```

```
{determine length of parameter;                          }
{sizeof function takes parameter as argument and returns}
{number of bytes in parameter                            }

   parms[5].pd_parmlen   := sizeof(DATELENGTH);

{value parameter}

   parms[5].pd_parm_type := parm_type_value;

{input parameter}

   parms[5].pd_io_type := [input_parm];


{call the Switch intrinsic to change modes}

   HPSWITCHTOCM (proc,  method,  nparms,  parms,  funclen,
                 funcptr, cond_code, status );

{test MPE XL status value and set ccode if not OK  }
{HPSETCCODE intrinsic used to pass on result of CM }
{call made by stub; calling program checks returned}
{ccode                                             }

   if status.all = all_ok then
      HPSETCCODE(cond_code)       {from CM proc          }
   else
      QUIT(status.info);          {Switch subsystem error}

end;   {Stub procedure CONVERTDATE}

BEGIN {dummy outer block}

END.  {end example 3-1B}
```

Note For a complete analysis of NM-to-CM Switch stub code, refer to Chapter 5.

Figure 3-3 illustrates how your CONVERTDATE Switch stub fits into the flow of control and enables you to access your CM SL CONVERTDATE procedure:

Figure F03-03 here.

Figure 3-3. NM—> CM Switch Summary 2

Example 3-2 implements a Switch stub for the `BINARY` intrinsic. This example illustrates a stub for a target procedure that has a functional return value:

Note This BINARY stub is only an example. The `BINARY` intrinsic is directly accessible from both Compatibility Mode and Native Mode, without the need of writing a stub to gain access.

Example 3-2. BINARY Intrinsic Switch Stub

```
$subprogram$
$standard_level 'EXT_MODCAL'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$
$os 'MPE/XL'$

PROGRAM XAMPL32(input, output);

CONST

   Pidt_Known          = 0; {by number}
   Pidt_Name           = 1; {by name}
   Pidt_Plabel         = 2; {by plabel}

   System_Sl           = 0; {search library}
   Logon_Pub_Sl        = 1;
   Logon_Group_Sl      = 2;
   Pub_Sl              = 3;
   Group_Sl            = 4;

   Method_Normal       = 0;  {Switch copy mode}
   Method_Split        = 1;
   Method_No_Copy      = 2;

   Parm_Type_Value     = 0; {value parameter}
   Parm_Type_Word_Ref  = 1; {reference parm, word addr}
   Parm_Type_Byte_Ref  = 2; {reference parm, byte addr}

   Ccg                 = 0; {condition code greater (>)}
   Ccl                 = 1; {condition code less (<)}
   Cce                   = 2; {condition code equal (=)}
   All_ok                = 0; {used in status check}

TYPE

   BIT8                  = 0..255;
   BIT16                 = 0..65535;
   BIT8_A1               = $ALIGNMENT 1$ BIT8;
   BIT16_A1              = $ALIGNMENT 1$ BIT16;
   CM_PROC_NAME          = PACKED ARRAY [1..16] of CHAR;
   GENERIC_BUFFER        = PACKED ARRAY [1..65535] of CHAR;
```

Example 3-2. BINARY Intrinsic Switch Stub, continued

```
   SCM_PROCEDURE =
      PACKED RECORD
      CASE p_proc_id_type : BIT8 of
         Pidt_Known:
            (p_fill     : BIT8_A1;
             p_proc_id  : BIT16_A1);

         Pidt_Name:
            (p_lib        : BIT8_A1;
            (p_proc_name  : CM_PROC_NAME);

         Pidt_Plabel:
            (p_plabel  : BIT16_A1);
      END;  {record}

   SCM_IO_TYPE = SET OF (input_parm, output_parm);

   PARM_DESC =
      PACKED RECORD
         pd_parmptr    : GLOBALANYPTR;
         pd_parmlen    : BIT16;
         pd_parm_type  : BIT16;
         pd_io_type    : SCM_IO_TYPE;
      END;  {record}
   SCM_PARM_DESC_ARRAY = ARRAY [0..31] of PARM_DESC;

   CCODE_TYPE          = shortint;

   XLSTATUS =
      RECORD
         CASE INTEGER OF
           0 :
               (all : INTEGER);
           1 :
               (info   : SHORTINT;
                subsys : SHORTINT);
      END;  {record}

   PROCEDURE HPSWITCHTOCM; INTRINSIC;

   PROCEDURE HPSETCCODE; INTRINSIC;

   PROCEDURE QUIT; INTRINSIC;

{End of OUTER BLOCK GLOBAL declarations}
```

Example 3-2. BINARY Intrinsic Switch Stub, continued

```
   FUNCTION BINARY $ALIAS 'BINARY'$
            (
       ANYVAR F1 : GENERIC_BUFFER;
              F2 : SHORTINT;
             ) : SHORTINT
          OPTION UNCHECKABLE_ANYVAR;

VAR

   proc               : SCM_PROCEDURE;
   parms              : SCM_PARM_DESC_ARRAY;
   method             : INTEGER;
   nparms             : INTEGER;
   funclen            : INTEGER;
   funcptr            : INTEGER;
   byte_len_of_parm   : BIT16;
   cond_code          : CCODE_TYPE;
   status             : XLSTATUS;

VAR loc_F2  : SHORTINT;
    funcval : SHORTINT;

begin {STUB procedure BINARY }

{ Initialization }

{ Setup procedure information -- name, lib, etc}

   proc.p_proc_id_type := Pidt_Name; {by name}
   proc.p_lib           := Pub_Sl;    {library}
   proc.p_proc_name     := 'BINARY          ';

{Setup misc. variables}

   method := Method_Normal;  {Switch copy mode}
   nparms := 2;
```

```
{Setup length/pointers for functional return if this }
{is a FUNCTION.  Set length to zero, pointer to NIL  }
{if this is not a FUNCTION.                          }

   funclen  := sizeof(funcval);  {A function}
   funcval := 0;
   funcptr  := INTEGER(LOCALANYPTR(ADDR(funcval)));

{Make a local copy of all VALUE parameters }

   loc_F2 := F2;
```

Example 3-2. BINARY Intrinsic Switch Stub, continued

```
{Build parameter descriptor array to describe each }
{parameter.                                        }

{F1 -- Input Only by Reference }

   byte_len_of_parm := F2 * 1;

   parms[0].pd_parmptr    := ADDR(F1);
   parms[0].pd_parmlen    := byte_len_of_parm;
   parms[0].pd_parm_type  := Parm_Type_Byte_Ref;
   parms[0].pd_io_type    := [Input_Parm];

{F2 -- Input Only by Value }

   byte_len_of_parm := 2;

   parms[1].pd_parmptr    := ADDR(loc_F2);
   parms[1].pd_parmlen    := byte_len_of_parm;
   parms[1].pd_parm_type  := Parm_Type_Value;
   parms[1].pd_io_type    := [Input_Parm];

{Do actual Switch call}

   HPSWITCHTOCM
      (proc,          {procedure info}
       method,        {Switch copy method}
       nparms,        {Number of parameters}
       parms,         {Parm descriptor array}
       funclen,       {Function return value length}
       funcptr,       {Address of functional return}
       cond_code,     {Condition code return}
       status);       {Switch status code}

   binary := 0;
   if (status.all <> all_ok) then
      BEGIN {Switch subsystem error}
         QUIT(status.info);
      END  {Switch subsystem error}
   else binary := funcval;

   HPSETCCODE(cond_code);

END;  {STUB procedure}

BEGIN {Program Outer Block code}

END. {Program Outer Block code}

{end Example 3-2}
```

Switch to CM Sequence

Examples 3-3 through 3-10 are a series of code examples, intended to illustrate the partial recompilation migration option, using an NM—> CM switch. In these examples, the calling routine migrates to Native Mode while the target procedure remains in Compatibility Mode.

The partial recompilation migration option typically begins with a program that calls procedures written in languages for which there is no NM compiler. Example 3-3 is a Pascal/V program that calls an SPL procedure.

Example 3-3. Pascal/V Program Calling SPL

```
PROGRAM EXAMPLE (input,output);

TYPE
   pac26 = packed array [1..26] of char;
   shortint = -32768..32767;

VAR
   buffer : pac26;
   count  : shortint;
   number : integer;

FUNCTION d2a(number : integer; VAR buffer : pac26) :
             shortint;
             external;

BEGIN
   number := 198765432;
   buffer := 'xxxxxxxxxxxxxxxxxxxxxxxxxx';
   count  := d2a(number, buffer);
   writeln('The number is: ',number);
   writeln('The buffer is: ',buffer);
   writeln('The count is: ',count);
END.

{end Example 3-3}
```

Example 3-4 illustrates a procedure that is written in a language for which Hewlett-Packard does not supply an NM compiler. It is the SPL procedure called by the Pascal/V program in Example 3-3.

Example 3-4. Called SPL Procedure

```
$CONTROL USLINIT,SUBPROGRAM,SEGMENT=EXAMPLE
BEGIN
   EQUATE BASE = 10;
   DEFINE D'MINIT = -214748256D#;
   EQUATE OFFSET = 48;
   EQUATE PLACES = 9;    << size of maxint, 32-bits >>

DOUBLE PROCEDURE D'EXP(X,Y);
   VALUE X, Y;
   INTEGER X, Y;
BEGIN
   DOUBLE XX;
<< D'EXP carries out exponentiation on positive 16-bit >>
<< integers, yielding a 32-bit result.  Negative       >>
<< arguments yield 0.                                  >>
   IF (X < 0) OR (Y < 0) THEN
      D'EXP := 0D
   ELSE
      BEGIN
         XX := 1D;
         WHILE (Y > 0) DO
            BEGIN
               XX := XX * DOUBLE(X);
               Y := Y - 1;
            END;
         D'EXP := XX;
      END;
END;

INTEGER PROCEDURE D2A(D'INT, ASCII'NUMERIC);
   VALUE D'INT;
   DOUBLE D'INT;
   BYTE ARRAY ASCII'NUMERIC;
```

```
<< D2A stands for Double-to-Ascii.  It converts a 32-bit>>
<< integer into an ASCII numeric in decimal            >>
<< representation and returns a count of the digits    >>
<< converted.  A "-" is placed at the beginning of the >>
<< number for negative integers.  The "-" sign         >>
<< character counts as a digit.                        >>
BEGIN
   DOUBLE SCALE;
   INTEGER DIGIT, INDEX, J;
   LOGICAL SKIP'ZEROS;
```

Example 3-4. Called SPL Procedure, continued

```
     INDEX := 0;                  << at left edge of buffer >>
     MOVE ASCII'NUMERIC(INDEX) := " ";
     MOVE ASCII'NUMERIC(INDEX+1) :=
          ASCII'NUMERIC(INDEX),(PLACES);
     IF (D'INT = D'MININT) THEN
        BEGIN
           MOVE ASCII'NUMERIC := "-214748256";
           INDEX := 10;
        END
     ELSE
        BEGIN
           IF (D'INT < 0D) THEN
              BEGIN
                 D'INT := \D'INT\;
                 ASCII'NUMERIC(INDEX) := "-";
                 INDEX := INDEX + 1;
              END;
           SKIP'ZEROS := TRUE;   << skip leading 0's >>
           FOR J := (PLACES - 1) STEP -1 UNTIL 0 DO
             BEGIN
               SCALE := D'EXP(BASE,J);
               DIGIT := INTEGER(D'INT/SCALE);
               IF (DIGIT <> 0) OR (SKIP'ZEROS = FALSE)
                 OR (J = 0) THEN
                  BEGIN
                    SKIP'ZEROS := FALSE;
                    D'INT := D'INT - (DOUBLE(DIGIT) * SCALE);
                    ASCII'NUMERIC(INDEX) :=
                           BYTE(DIGIT + OFFSET);
                    INDEX := INDEX + 1;
                  END;
              END;
        END;
     D2A := INDEX; << return how many characters in number>>
  END;

  END.
```

```
{end Example 3-4}
```

Before migration, the Segmenter was needed to combine the preceding program and its called procedure. Example 3-5 is an illustration of such a Segmenter session to combine the program and the procedure.

Example 3-5. Segmenter Session

```
:SPL EXAMPLE2,,*LP

$CONTROL USLINIT,SUBPROGRAM,SEGMENT=EXAMPLE

                          ^

***** WARNING #001 *****  SUBPROGRAM AND USL INITIALIZED

 PRIMARY DB STORAGE=%000;    SECONDARY DB STORAGE=%00000
 NO. ERRORS=0000;            NO. WARNINGS=0001
 PROCESSOR TIME=0:00:00;     ELAPSED TIME=0:00:00

END OF COMPILE
:SEGMENTER
HP32050A.02.00  SEGMENTER/3000 (C) HEWLETT-PACKARD CO 1985
-SL SL
-PURGESL SEGMENT,EXAMPLE
-USL $OLDPASS
-LISTUSL

USL FILE $OLDPASS.DTJ.DTEST

EXAMPLE
   D2A               143 P   A C N R
   D'EXP              34 P   A C N R

FILE SIZE   144000(  620.  0)
DIR. USED      255(    1. 55) INFO USED      205(    1.  5)
DIR. GARB.       0(    0.  0) INFO GARB.       0(    0.  0)
DIR. AVAIL.  14123(   60.123) INFO AVAIL. 127173(  534.173)
-ADDSL EXAMPLE
-LISTSL EXAMPLE
```

```
SL FILE SL.DTJ.DTEST

SEGMENT   2 EXAMPLE         LENGTH  204

   ENTRY POINTS    CHECK CAL STT ADR
   D2A               0    C    1    0
   D'EXP             0    C    2  143

   EXTERNALS       CHECK STT SEG

001
-EXIT

END OF SUBSYSTEM

{end Example 3-5}
```

The migration of routines for which NM compilers exist can be accomplished by recompiling the routine using one of the NM optimizing compilers. The code example in Example 3-6 is the HP Pascal/XL version of the Pascal/V program in Example 3-3. It illustrates that migration through recompilation may involve little or no change to source code.

Example 3-6. HP Pascal/XL Version of Pascal/V Program

```
$standard_level 'ext_modcal'$
$os 'MPE/XL'$
PROGRAM XAMPL36(input,output);

TYPE
   pac26 = packed array [1..26] of char;
   shortint = -32768..32767;

VAR
   buffer : pac26;
   count  : shortint;
   number : integer;

FUNCTION d2a(number : integer; VAR buffer : pac26) :
              shortint;
              external;

BEGIN
   number := 198765432;
   buffer := 'xxxxxxxxxxxxxxxxxxxxxxxxxx';
   count  := d2a(number, buffer);
   writeln('The number is: ', number);
   writeln('The buffer is: ', buffer);
   writeln('The count is: ', count);
END.

{end Example 3-6}
```

However, if nothing else is done, an error results because the NM loader cannot resolve the external procedure reference because the procedure resides in a CM SL. Example 3-7 illustrates the link error obtained from referencing the called SPL procedure from the HP Pascal/XL program in Example 3-6.

Example 3-7. Link Error

```
:PASXLLK XAMPL36,,$null
PAGE  1 HP PASCAL/XL HP31502A.01.00  COPYRIGHT HEWLETT-PACKARD CO. 1986

        SAT, NOV 21, 1987, 11:40 AM

                   NUMBER OF ERRORS =  0    NUMBER OF WARNINGS =  0
                   PROCESSOR TIME  0: 0: 1  ELAPSED TIME  0: 0: 0
                   NUMBER OF LINES =    25  LINES/MINUTE =   2645.5
                   NUMBER OF NOTES =  0
END OF COMPILE
LinkEd> link from $oldpass;to=
HP Link Editor/XL (HP30315A.01.00)   Copyright Hewlett-Packard Co 1986


END OF LINK
:$oldpass
UNRESOLVED EXTERNALS: d2a  (LDRERR 512)
UNABLE TO LOAD PROGRAM TO BE RUN.  (CIERR 625)
```

There are two alternatives for resolving the external reference so that your NM routine can access the CM SPL procedure:

- You can alter the original source code of the HP Pascal/XL routine to include the required data setup and an in-line Switch that calls the appropriate Switch intrinsic. Example 3-8 illustrates such an in-line Switch.
- You can potentially avoid any changes to original source code by writing a Switch stub to perform the actual mode switch (by setting up parameters for and invoking the appropriate Switch intrinsic). Example 3-9 illustrates a Switch stub.

Example 3-8. HP Pascal/XL Program With In-line Switch

```
$standard_level 'ext_modcal'$
$os 'MPE/XL'$
PROGRAM XAMPL38(INPUT, OUTPUT);

CONST

   Pidt_Known          = 0; {by number}
   Pidt_Name           = 1; {by name}
   Pidt_Plabel         = 2; {by plabel}

   System_Sl           = 0; {search library}
   Logon_Pub_Sl        = 1;
   Logon_Group_Sl      = 2;
   Pub_Sl              = 3;
   Group_Sl            = 4;

   Method_Normal       = 0;  {Switch copy mode}
   Method_Split        = 1;
   Method_No_Copy      = 2;

   Parm_Type_Value     = 0; {value parameter}
   Parm_Type_Word_Ref  = 1; {reference parm, word addr}
   Parm_Type_Byte_Ref  = 2; {reference parm, byte addr}

   Ccg                 = 0; {condition code greater (>)}
   Ccl                 = 1; {condition code less (<)}
```

```
Cce                   = 2; {condition code equal (=)}
All_ok                = 0; {used in status check}
```

Example 3-8. HP Pascal/XL Program With Inline Switch, continued

```
TYPE
   BIT8                 = 0..255;
   BIT16                = 0..65535;
   BIT8_A1              = $ALIGNMENT 1$ BIT8;
   BIT16_A1             = $ALIGNMENT 1$ BIT16;
   CM_PROC_NAME         = PACKED ARRAY [1..16] of CHAR;
   GENERIC_BUFFER       = PACKED ARRAY [1..65535] of CHAR;
   SHORTINT             = -32768..32767;

   SCM_PROCEDURE =
      PACKED RECORD
      CASE p_proc_id_type : BIT8 of
         Pidt_Known:
            (p_fill     : BIT8_A1;
             p_proc_id  : BIT16_A1);

         Pidt_Name:
            (p_lib        : BIT8_A1;
            (p_proc_name  : CM_PROC_NAME);

         Pidt_Plabel:
            (p_plabel  : BIT16_A1);
      END;  {record}

   SCM_IO_TYPE = SET OF (input_parm, output_parm);

   PARM_DESC = PACKED RECORD
         pd_parmptr    : GLOBALANYPTR;
         pd_parmlen    : BIT16;
         pd_parm_type  : BIT16;
         pd_io_type    : SCM_IO_TYPE;
      END;  {record}

   SCM_PARM_DESC_ARRAY = ARRAY [0..31] of PARM_DESC;

   CCODE_TYPE           = shortint;
```

```
      xlstatus = record case integer of
         0 : (all : integer);
         1 : (info : shortint;
              subsys : shortint);
      end;
      pac26 = packed array [1..26] of char;
      localanyptrrec = record
         rtn_addr : localanyptr;   { for alignment }
      end;

   VAR
      condcode    : shortint;
      count       : shortint;
      status      : xlstatus;
      parms       : scm_parm_desc_array;
      proc_info   : scm_procedure;
      i           : integer;
```

Example 3-8. HP Pascal/XL Program With Inline Switch, continued

```
   VAR
      method      : integer;
      buffer      : pac26;
      nparms      : integer;
      rtn_len     : integer;
      rtn_val     : localanyptrrec;   {localanyptr}

   procedure HPSWITCHTOCM; intrinsic;

   BEGIN
      proc_info.p_lib := group_sl;
      proc_info.p_proc_name := 'D2A ';  {procedure name}

      method := method_normal;          {single stack}

      nparms := 2;

      i := 123456789;               {value to convert}
```

```
    parms[1].pd_parmptr := addr(i);
    parms[1].pd_parmlen := 4;         {bytes}
    parms[1].pd_parm_type := parm_type_value;
    parms[1].pd_io_type :=
             parms[1].pd_io_type + [input_parm];

    buffer := '';

    parms[2].pd_parmptr := addr(buffer);
    parms[2].pd_parmlen := 6;           {bytes}
    parms[2].pd_parm_type := parm_type_byte_ref;
    parms[2].pd_io_type :=
             parms[2].pd_io_type + [output_parm];

    rtn_len := 2;        {bytes}
    rtn_val.rtn_addr := addr(count);

    condcode := 2;       {CCE}
    status.all := 0;

    writeln('The number to convert = ',i);
    writeln('The buffer contents are: ',buffer);

    HPSWITCHTOCM(proc_info, method, nparms, parms,
                 rtn_len, rtn_val, condcode, status);

    writeln('Switch status : subsys ',status.subsys);
    writeln('              :   info ',status.info);
    writeln('Count of converted digits (+sign) = ',count);
    writeln('The buffer contents are: ',buffer);

 END.  {end Example 3-8}
```

Note

The *status* **parameter** returns information concerning the success of the intrinsic's execution. It is considered good programming practice to check the value of *status* after making an intrinsic call. Normally this would involve a conditional statement, with the value of *status* determining the execution path. In Example 3-8, this sort of test is intentionally omitted. Instead, the routine prints out the values of the *status* parameter fields.

Example 3-9. HP Pascal/XL Switch Stub

```
$subprogram$
$check_actual_parm 0$
$check_formal_parm 0$
$standard_level 'ext_modcal'$
$os 'MPE/XL'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$

PROGRAM XAMPL39(INPUT, OUTPUT);

CONST

   Pidt_Known         = 0; {by number}
   Pidt_Name          = 1; {by name}
   Pidt_Plabel        = 2; {by plabel}

   System_Sl          = 0; {search library}
   Logon_Pub_Sl       = 1;
   Logon_Group_Sl     = 2;
   Pub_Sl             = 3;
   Group_Sl           = 4;

   Method_Normal      = 0;  {Switch copy mode}
   Method_Split       = 1;
   Method_No_Copy     = 2;

   Parm_Type_Value    = 0; {value parameter}
   Parm_Type_Word_Ref = 1; {reference parm, word addr}
   Parm_Type_Byte_Ref = 2; {reference parm, byte addr}
   Ccg                 = 0; {condition code greater (>)}
   Ccl                 = 1; {condition code less (<)}
   Cce                 = 2; {condition code equal (=)}
   All_ok              = 0; {used in status check}
```

Example 3-9. HP Pascal/XL Switch Stub, continued

```
TYPE
   BIT8                = 0..255;
   BIT16               = 0..65535;
   BIT8_A1             = $ALIGNMENT 1$ BIT8;
   BIT16_A1            = $ALIGNMENT 1$ BIT16;
   CM_PROC_NAME        = PACKED ARRAY [1..16] of CHAR;
   GENERIC_BUFFER      = PACKED ARRAY [1..65535] of CHAR;

   SCM_PROCEDURE =
      PACKED RECORD
      CASE p_proc_id_type : BIT8 of
         Pidt_Known:
            (p_fill     : BIT8_A1;
             p_proc_id  : BIT16_A1);

         Pidt_Name:
            (p_lib        : BIT8_A1;
            (p_proc_name  : CM_PROC_NAME);

         Pidt_Plabel:
            (p_plabel  : BIT16_A1);
      END;  {record}

   SCM_IO_TYPE = SET OF (input_parm, output_parm);

   PARM_DESC = PACKED RECORD
         pd_parmptr    : GLOBALANYPTR;
         pd_parmlen    : BIT16;
         pd_parm_type  : BIT16;
         pd_io_type    : SCM_IO_TYPE;
      END;  {record}
      SCM_PARM_DESC_ARRAY = ARRAY [0..31] of PARM_DESC;

      CCODE_TYPE          = shortint;

      xlstatus = record case integer of
         0 : (all : integer);
         1 : (info : shortint;
              subsys : shortint);
      end;

   {Declare all types which can be passed to this stub }
   {so that 16-bit alignments are allowed.             }

      hp_shortint  =  $alignment 2$ shortint;
      hp_integer   =  $alignment 2$ integer;
      hp_real      =  $alignment 2$ real;
      hp_long      =  $alignment 2$ longreal;
      hp_char      =  $alignment 1$ char;
```

Example 3-9. HP Pascal/XL Switch Stub, continued

```
   procedure HPSWITCHTOCM; intrinsic;

   procedure HPSETCCODE; intrinsic;

   procedure QUIT; intrinsic;

   {End of OUTER BLOCK GLOBAL declarations }

   FUNCTION D2A $ALIAS 'D2A'$
                 (
                         NUMBER : INTEGER;
                  ANYVAR BUFFER : GENERIC_BUFFER
                 ) : SHORTINT
                 OPTION UNCHECKABLE_ANYVAR;

   VAR
      proc_info   : scm_procedure;
      parms       : scm_parm_desc_array;
      method      : integer;
      nparms      : integer;
      funclen     : integer;
      funcptr     : integer;
      byte_len_of_parm : bit16;
      condcode    : shortint;
      status      : xlstatus;

   VAR retval  :  shortint;
   VAR loc_number  :  integer;

   BEGIN  {STUB procedure D2A}

   {Initialization}

   {Set up procedure information--name, lib, etc.}

      proc_info.p_proc_id_type := pidt_load;
      proc_info.p_lib := group_sl;
      proc_info.p_proc_name := 'D2A ';  {procedure name}

   {Set up miscellaneous variables}

      method := method_normal;          {single stack}
      nparms := 2;

   {Set up length/pointers for functional return if this}
   {is a FUNCTION.  Set length to zero, pointer to NIL  }
   {if this is not a FUNCTION.                          }

      funclen := SIZEOF(retval);
      funcptr := INTEGER(LOCALANYPTR(ADDR(retval)));
```

Example 3-9. HP Pascal/XL Switch Stub, continued

```
   {Make a local copy of all VALUE parameters}

      loc_number := NUMBER;
```

```
{Build the parm descriptor array to describe each}
{parameter.                                      }

{NUMBER -- Input only by VALUE}

   byte_len_of_parm := 4;

   parms[0].pd_parmptr := addr(loc_number);
   parms[0].pd_parmlen := byte_len_of_parm;
   parms[0].pd_parm_type := parm_type_value;
   parms[0].pd_io_type := [input_parm];

{BUFFER -- Output only by REFERENCE}

   byte_len_of_parm := 6;

   parms[1].pd_parmptr = addr(buffer);
   parms[1].pd_parmlen = byte_len_of_parm;
   parms[1].pd_parm_type := parm_type_byte_ref;
   parms[1].pd_io_type := [output_parm];

{Do Switch call}

   HPSWITCHTOCM(proc_info, method, nparms, parms,
                funclen, funcptr, condcode, status);

   if (status.all <> all_ok) then
      begin  {Switch subsystem error}
         QUIT (status.info);  {handles error codes}
      end;  {Switch subsystem error}

   HPSETCCODE (condcode);

   D2A := retval;

END;  {stub procedure D2A}

begin  {main}
```

```
end.  {main}

{end Example 3-9}
```

On MPE XL, a LinkEdit session is used to combine a routine and its external references. Example 3-10 represents a LinkEdit session to combine the compiled stub procedure of Example 3-9 with the compiled HP Pascal/XL program of Example 3-6.

When the LinkEdit session is completed, the partial recompilation migration is complete. The error of Example 3-7 will no longer occur, and you can run the HP Pascal/XL routine, still accessing the CM SPL procedure.

Before beginning the LinkEdit session, you need to compile both the source for the program in Example 3-6 and the Switch stub source code in Example 3-9. Since both routines are written in Pascal, use the `PASXL` command to invoke the HP Pascal/XL compiler and create NM object modules.

The following command compiles the main program (Example 3-6) and yields an object code file named `APPOBJ`:

```
PASXL XAMPL36, APPOBJ
```

The next command compiles the Switch stub (Example 3-9) and yields an object code file named `STUBOBJ`:

```
PASXL XAMPL39, STUBOBJ
```

For more information about the `PASXL` command or the commands to invoke other NM compilers, refer to the *MPE XL Commands Reference Manual* (32650-90003).

Example 3-10. LinkEdit Session

```
{The first LinkEdit session illustrates how to
 create or designate an NM Executable Library (XL)
 and add a Switch stub object module to it.  This
 example also shows how to create an executable
 NM program that accesses the XL.  If you want to
 make the Switch stub object module accessible to
 more than one application, this is the appropriate
 course to follow.                                 }

    :LINKEDIT

 {If the XL to which the stub object code is to be
  added does not exist, create it and add the object
  module to it as follows.                         }

    LinkEd> BUILDXL MYXL
    LinkEd> ADDXL FROM=STUBOBJ

{If the XL to which the stub object code is to be
 added already exists, designate it and add the
 object module to it as follows.                 }

    LinkEd> XL XL=MYXL
    LinkEd> ADDXL FROM=STUBOBJ

{Finally, to create an executable program (MYPROG),
 use the LINK command as follows:                       }

    LinkEd> LINK FROM=APPOBJ; TO=MYPROG; XL=MYXL


{The second LinkEdit session illustrates how to
 link a Switch stub object module into an NM program
 file.  If you do not want to make the object code
 available for shared use, this is the appropriate
 choice.                                            }
```

```
        :LINKEDIT

        LinkEd> LINK FROM=APPOBJ, STUBOBJ; TO=MYPROG

   {You can also issue the LINK command directly at
    the Command Interpreter prompt (:), as follows:  }

        :LINK FROM=APPOBJ, STUBOBJ; TO=MYPROG
```

For more information about the Link Editor, refer to the *Link Editor/XL Reference Manual* (32650-90030).

Special Considerations and Restrictions

There are certain considerations to bear in mind when you use Switch.

General

General considerations apply no matter what the direction of the mixed-mode call:

- Mixed-mode procedure calls require an increased level of programming complexity.
- The overhead of Switch can be significant.
- Because Switch does no probing on addresses passed by the caller, data memory protection faults are likely when you pass in bad parameter addresses or lengths.
- Since Switch does no parameter type or alignment checking, the target procedure must make the necessary checks.

The following considerations apply to a particular direction of mixed-mode call.

NM—> CM

When making mixed-mode procedure calls in the direction NM—> CM, the following considerations apply:

- Any CM procedure called from NM code must reside in a CM SL. The SL containing the procedure must be accessible via the `LOADPROC` or `HPLOADCMPROCEDURE` intrinsic. This may be the system library, a group library, or a user library.
- Unlike other MPE XL intrinsics, the Switch intrinsics do not promote their privilege level. If your user code calls Switch to access a non-Privileged target procedure, the execution level remains unchanged. On the other hand, if your user code calls Switch to access a Privileged target, the execution level becomes Privileged after the target is accessed.
- HP Pascal/XL does not support the equivalent of SPL's option variable, and `HPSWITCHTOCM` knows nothing about such procedures. Option variable procedures are implemented in Compatibility Mode by pushing a 16- or 32-bit mask on the CM stack immediately before calling the procedure.

  You can use the `HPSWITCHTOCM` intrinsic to call an option variable procedure, but it is your responsibility to construct the mask (refer to the *SPL Reference Manual* (30000-90024) for more information on the `OPTION VARIABLE` option) and to describe it to `HPSWITCHTOCM` as the first parameter. See Example 3-11 for an illustration of the use of `HPSWITCHTOCM` to call an option variable procedure.

Example 3-11. Stub for Option Variable Target Procedure

```
$subprogram$
$standard_level 'ext_modcal'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$

PROGRAM XAMPL311(input, output);

  CONST
    Pidt_Known          = 0;  { By number }
    Pidt_Name           = 1;  { By name   }
    Pidt_Plabel         = 2;  { By PLABEL }

    System_Sl           = 0;
    Logon_Pub_Sl        = 1;
    Logon_Group_Sl      = 2;
    Pub_Sl              = 3;
    Group_Sl            = 4;

    Method_Normal   = 0;  { Not callable from split stack}
    Method_Split    = 1;  { Callable in split stack mode }
    Method_No_Copy  = 2;  { No-copy method               }

    Parm_Type_Value    = 0;  { value parameter }
    Parm_Type_Word_Ref = 1;  { reference parm, word addr }
    Parm_Type_Byte_Ref = 2;  { reference parm, byte addr }

    Ccg              = 0;  { condition code greater (>) }
    Ccl              = 1;  { condition code less    (<) }
    Cce              = 2;  { condition code equal   (=) }
    All_Ok           = 0;  { Used in status check       }

  TYPE
    BIT8              = 0..255;
    BIT16             = 0..65535;
    BIT8_A1           = $ALIGNMENT 1$ BIT8;
```

```
   BIT16_A1          = $ALIGNMENT 1$ BIT16;
   CM_PROC_NAME      = PACKED ARRAY [1..16] OF CHAR;
   GENERIC_BUFFER    = PACKED ARRAY [1..65535] OF CHAR;

   SCM_PROCEDURE        =
     PACKED RECORD
     CASE p_proc_id_type : BIT8 OF
       Pidt_Known: (p_fill    : BIT8_A1;
                    p_proc_id : BIT16_A1);
       Pidt_Name: (p_lib       : BIT8_A1;
                   p_proc_name : CM_PROC_NAME);
       Pidt_Plabel: (p_plabel : BIT16_A1);
     END;  { record }
```

Example 3-11. Stub for Option Variable Target Procedure, continued

```
   SCM_IO_TYPE      = SET OF (input_parm, output_parm);

   PARM_DESC            =
     PACKED RECORD
     pd_parmptr         : GLOBALANYPTR;
     pd_parmlen         : BIT16;
     pd_parm_type       : BIT16;
     pd_io_type         : SCM_IO_TYPE;
     END;

   SCM_PARM_DESC_ARRAY  = ARRAY [0..31] OF PARM_DESC;

   CCODE_TYPE           = shortint;

   XLSTATUS             =
     RECORD
     CASE INTEGER OF
       0: (all : INTEGER);
       1: (info   : SHORTINT;
           subsys : SHORTINT);
     END;  { record }

 PROCEDURE HPSWITCHTOCM; INTRINSIC;
```

```
     PROCEDURE HPSETCCODE; INTRINSIC;

     PROCEDURE QUIT; INTRINSIC;

   { End of OUTER BLOCK GLOBAL declarations }

    PROCEDURE FCHECK  $ALIAS 'FCHECK'$
              (
              FILENUM : SHORTINT;
          VAR ERRORCODE : SHORTINT;
          VAR TLOG : SHORTINT;
          VAR BLKNUM : INTEGER;
          VAR NUMRECS : SHORTINT
               );

       VAR
         proc               : SCM_PROCEDURE;
         parms              : SCM_PARM_DESC_ARRAY;
         method             : INTEGER;
         nparms             : INTEGER;
         funclen            : INTEGER;
         funcptr            : INTEGER;
         byte_len_of_parm   : BIT16;
         cond_code          : CCODE_TYPE;
         status             : XLSTATUS;
         mask               : BIT16;
```

Example 3-11 Stub for Option Variable Target Procedure, continued

```
    VAR loc_FILENUM : SHORTINT;

    begin { STUB procedure FCHECK }


     { Initialization }


     { Setup procedure information--name, lib, etc. }
```

```
proc.p_proc_id_type :=  Pidt_Name; { By name }
proc.p_lib          :=  System_Sl; { Library }
proc.p_proc_name    :=  'FCHECK          ';

{ Setup misc. variables }

mask              :=  0;
method            :=  Method_Normal;  { Split stack? }
nparms            :=  6;              { 5 + mask }

{ Setup length/pointers for functional return if this }
{ is a FUNCTION.  Set length to zero, pointer to NIL  }
{ if this is not a FUNCTION.                          }

funclen           :=  0;    { Not a function }
funcptr           :=  0;

{ Make a local copy of all VALUE parameters }

loc_FILENUM := FILENUM;

{ Build parameter descriptor array to describe each }
{ parameter.                                        }


{ FILENUM -- Input Only by VALUE }

byte_len_of_parm := 2;

mask := binary('10000');
parms[0].pd_parmptr   := ADDR(loc_FILENUM);
parms[0].pd_parmlen   := byte_len_of_parm;
parms[0].pd_parm_type := Parm_Type_Value;
parms[0].pd_io_type   := [Input_Parm];

{ ERRORCODE -- Output Only by REFERENCE }
```

```
     byte_len_of_parm := 2;

     mask := ior16(mask, binary('01000'));
     parms[1].pd_parmptr   := ADDR(ERRORCODE);
     parms[1].pd_parmlen   := byte_len_of_parm;
     parms[1].pd_parm_type := Parm_Type_Word_Ref;
     parms[1].pd_io_type   := [Output_Parm];
```

Example 3-11. Stub for Option Variable Target Procedure, continued

```
     { TLOG -- Output Only by REFERENCE }

     byte_len_of_parm := 2;

     mask := ior16(mask, binary('00100'));
     parms[2].pd_parmptr   := ADDR(TLOG);
     parms[2].pd_parmlen   := byte_len_of_parm;
     parms[2].pd_parm_type := Parm_Type_Word_Ref;
     parms[2].pd_io_type   := [Output_Parm];

     { BLKNUM -- Output Only by REFERENCE }

     byte_len_of_parm := 4;

     mask := ior16(mask, binary('00010'));
     parms[3].pd_parmptr   := ADDR(BLKNUM);
     parms[3].pd_parmlen   := byte_len_of_parm;
     parms[3].pd_parm_type := Parm_Type_Word_Ref;
     parms[3].pd_io_type   := [Output_Parm];

     { NUMRECS -- Output Only by REFERENCE }

     byte_len_of_parm := 2;

     mask := ior16(mask, binary('00001'));
     parms[4].pd_parmptr   := ADDR(NUMRECS);
     parms[4].pd_parmlen   := byte_len_of_parm;
     parms[4].pd_parm_type := Parm_Type_Word_Ref;
     parms[4].pd_io_type   := [Output_Parm];
```

```
    { mask parameter }

    parms[5].pd_parmptr   := ADDR(mask);
    parms[5].pd_parmlen  := byte_len_of_parm;
    parms[5].pd_parm_type := parm_type_value;
    parms[5].pd_io_type  := [Input_Parm];

    { Do the actual SWITCH call }

    HPSWITCHTOCM(proc,         { Procedure info         }
                 method,       { Split stack ?          }
                 nparms,       { Number of parameters   }
                 parms,        { Parm descriptor array  }
                 funclen,      { func ret value length  }
                 funcptr,      { Addr of func return    }
                 cond_code,    { cond. code return      }
                 status);      { SWITCH status code     }

    if (status.all <> all_ok) then
      BEGIN { SWITCH subsystem error }
         QUIT(status.info);
      END;  { SWITCH subsystem error }
```

Example 3-11. Stub for Option Variable Target Procedure, continued

```
    HPSETCCODE(cond_code);

    end; { STUB procedure FCHECK }

    BEGIN { Program Outer block code }

    END.  { Program Outer block code }

  {end Example 3-11}
```

Testing and Debugging Considerations

You must test Switch stubs and inline Switches in the run-time environment on an MPE XL-based system. You must either compile the stub and install the compiled version in the appropriate system library or merge it with the code requiring the Switch call and recompile that merged source. You can also compile the stub and then link it with the USL/SOM file for the program that calls it. Before you can verify correct execution of the stub, the target procedure that Switch is to invoke must be present in the appropriate callable library of the other mode. You can use the MPE XL debug facilities to troubleshoot, should you encounter problems.

The following are testing and debugging considerations that apply specifically to switches in the NM—> CM direction:

- Verify that the other-mode procedure operates correctly by calling it from its own mode.
- Verify that the switch worked correctly by checking its *status* parameter return. It should be all zeros.
- Verify that the method and number of parameters of the target procedure are correctly specified.
- Verify that the parameter types in the **parms** descriptor array are correct.
- Verify that the parameters for the function return length and function return value were correctly specified (if applicable).
- Verify that name lengths are correct.
- Verify that `HPLOADCMPROCEDURE` succeeded (by checking the *status* parameter return) before calling `HPSWITCHTOCM`.
- Insure that array lengths specified in the **parms** descriptor array are not longer than the actual arrays being passed. If you must specify an array length longer than the actual length, then make sure the array is an input/output parameter, rather than output alone.

4

CM-to-NM Procedure Calls

This section presents the following topics:

- Review of the flow of control in CM—> NM switches
- Details of CM—> NM switches, including their inner workings, syntax, parameters, and examples
- Special considerations and restrictions that apply to mixed-mode procedure calls
- Testing and debugging considerations

This chapter provides the detailed reference information you will need if you write your own CM-to-NM Switch stubs. Refer to Chapter 5 for further information on writing Switch stubs.

Overview

This overview presents mixed-mode procedure calls in the CM—> NM direction in two ways:

- Schematic flow-of-control diagram
- Stepwise sequence of events

Flow of Control: CM—> NM

The flow of control for mixed-mode procedure calls in the direction CM—> NM is illustrated in Figure 4-1:

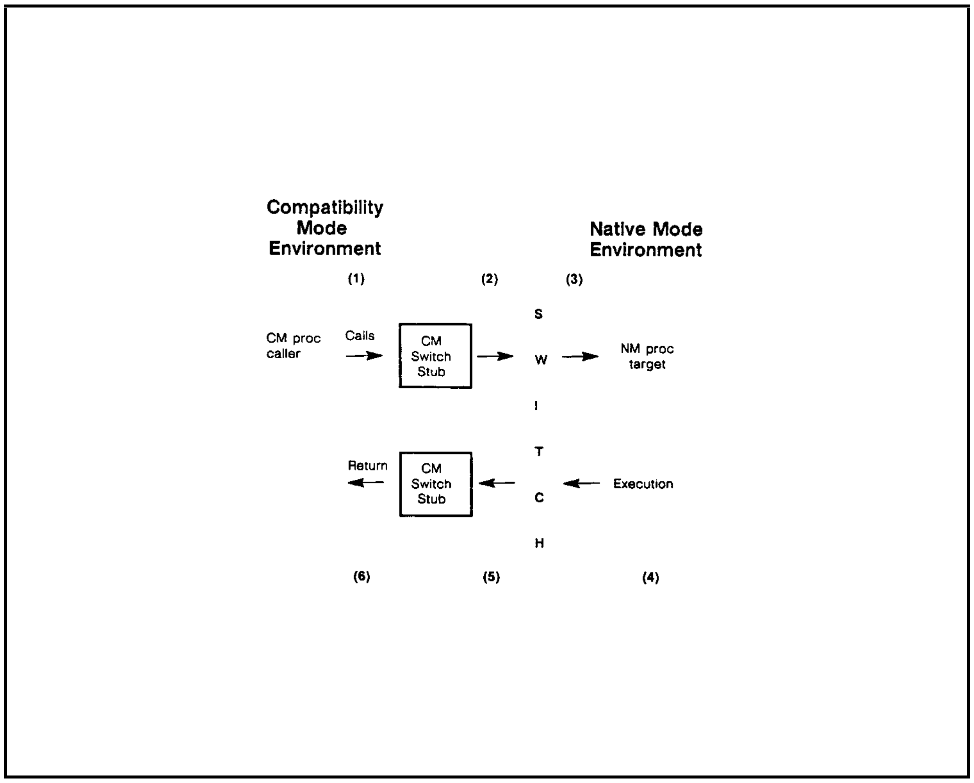


Figure 4-1. CM—> NM Switch Summary 1

Stepwise Switch to NM

A mixed-mode procedure call in the direction CM—> NM involves the following steps, as indicated in Figure 4-1:

1 The CM code needing to access an NM routine calls Switch, either by means of in-line code or a Switch stub.

2 The in-line code or CM Switch stub sets up the data structures and parameters required by Switch and makes a call to either the `HPSWTONMNAME` intrinsic or the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics.

3 `HPSWTONMNAME` or `HPSWTONMPLABEL` calls the NM routine, after preparing the NM registers as though the call was being made from Native Mode.

4 The NM routine executes as though called from Native Mode and returns the functional return value (if any) to the calling Switch intrinsic. Reference parameters are modified by the NM routine.

5 If the NM routine has a functional return value, `HPSWTONMNAME` or `HPSWTONMPLABEL` copies the value back to the CM stack.

6 The stub or in-line code checks whether the Switch operation was successful and then control returns to the CM routine.

Note `HPSWTONMNAME` or a combination of `HPLOADNMPROC` and `HPSWTONMPLABEL` are the system intrinsics called to pass CM values and parameters to Native Mode, to change the execution mode, and to invoke an NM routine. `HPSWTONMNAME`, `HPLOADNMPROC`, and `HPSWTONMPLABEL` reside in the CM system library, `SL.PUB.SYS`. The only difference between using `HPSWTONMNAME` and `HPSWTONMPLABEL` is the manner in which the target routine is identified.

Switch to NM Details

Switch requires the following information to call an NM procedure from a CM procedure:

- Name of the procedure or an NM plabel (32-bits)
- If the NM procedure is specified by name, the following additional items are required:
  - Length of the procedure name
  - NM library to search for the target procedure
  - Length of the library name
- Number of parameters being passed to the NM procedure
- Array of values and/or pointers to the parameters being passed
- Array of descriptions of the parameters being passed
- Space reserved at the beginning of the parameter list for a function return value, if calling a function

The CM—> NM Switch must make the parameters of the calling CM procedure understandable to the NM procedure being called and must also make the data value(s) and status returned by the NM procedure understandable to the CM procedure.

The mechanism that Switch uses to enable mixed-mode calls in the CM—> NM direction are the `HPSWTONMNAME`, `HPSWTONMPLABEL`, and `HPLOADNMPROC` intrinsics. These intrinsics do address translation and copy parameters, as needed.

These intrinsics differ only in the way they identify the target procedure. `HPSWTONMNAME` uses the DB-relative address of a byte array passed by reference to specify the target procedure, while `HPSWTONMPLABEL` uses a 32-bit NM plabel (obtained from `HPLOADNMPROC`) to make this specification. With `HPSWTONMNAME`, the NM loader converts the procedure name to a plabel by means of a hashing function. `HPSWTONMPLABEL` eliminates the overhead of this name-to-plabel mapping.

Note The first time a switch by name occurs in either direction, a load from the library occurs. Thereafter, Switch uses an internal hash table to quickly invoke the already loaded procedure.

HPSWTONM NAME Intrinsic

The `HPSWTONMNAME` intrinsic allows CM user programs, user libraries, and system code to invoke NM procedures. In a manner similar to the `HPSWITCHTOCM` intrinsic, `HPSWTONMNAME` does the following:

- Converts the CM references in the argument list to virtual NM addresses
- Changes the execution mode
- Invokes the NM procedure specified by the CM caller

Since NM code can address the entire CM stack, there is no copying of reference parameters. The length of each actual parameter is not needed because lengths are implied in the descriptor list.

The syntax of this intrinsic and detailed explanations of its parameters are given in the following paragraphs. Also provided are examples of switches in the CM—> NM direction.

Note Switches by name involve high system overhead on the first call per name, but substantially lower overhead on each subsequent call for that name. The `HPSWTONMNAME`, `HPSWITCHTOCM`, `HPLOADCMPROCEDURE`, and `HPLOADNMPROC` intrinsics perform a hashing function on the name of the other-mode procedure and store the plabel for that procedure in a system internal hash table. The `LOADPROC` intrinsic, on the other hand, does not perform any hashing and, consequently, involves high system overhead every time it is called.

Syntax

Prior to calling the HPSWTONMNAME intrinsic, your programming language may require you to declare it. In Pascal/V, the declaration is as follows:

```
        FUNCTION HPSWTONMNAME : INTEGER; INTRINSIC;
```

Next comes an example of a Pascal/V call to this intrinsic:

```
        return_status := HPSWTONMNAME (procname, proclen,
                         libname, liblen, nparms, arglist,
                         argdesc, functype);
```

You call the HPSWTONMNAME intrinsic with eight parameters. These parameters provide Switch with the following information:

- Name of the NM routine
- Length of the procedure name
- NM library to search for the target procedure
- Length of the library name
- Number of parameters of the NM routine
- Parameter list
- Parameter description list
- Type of the functional return value (if any)

All this information is key to the correct operation of mixed-mode procedure calls.

Caution The strings you supply as values of the **Procname** and **libname** parameters must exactly match the names of the target NM routine and its NM library, respectively.

Parameters

A detailed explanation of the parameters of the `HPSWTONMNAME` intrinsics follows.

Required parameters are shown in **boldface**; optional parameters are shown in *italics*.

procname **byte array by reference (required)**

Passes the target procedure name. The target procedure must be contained in an Executable Library (XL). If the value of **procname** is invalid, blank, or does not contain your NM procedure, `NL.PUB.SYS` is searched.

proclen **16-bit signed integer by value (required)**

Passes the byte length of the procedure name.

libname **byte array by reference (required)**

Passes the name of the NM library to be searched for the target. If the value of **libname** is invalid, blank or does not contain your NM procedure, `NL.PUB.SYS` is searched.

liblen **16-bit signed integer by value (required)**

Passes the byte length of the library name.

nparms **16-bit signed integer by value (required)**

Passes the number of parameters you are passing to the target NM procedure. It specifies the length of the **argdesc** array. You must also account for any hidden parameters due to `ANYVAR` parameters, an extensible parameter list, and so forth. For more information on hidden parameters, refer to the *HP Pascal/XL Reference Manual* (31502-90002).

arglist **16-bit signed integer array by reference (required)**

Passes the actual parameters you are going to pass to the NM procedure.

argdesc **16-bit signed integer array by reference (required)**

Passes integer codes describing the parameters held in the **arglist** array. Refer to Tables 4-2, 4-3, and 4-4.

functype **16-bit signed integer by value (required)**

Passes the data type of the value the target procedure returns if it is a function. If the target is not a function, the value of this parameter is zero. The supported function types are described in Table 4-5.

Note It is your responsibility to build and load the argument arrays and make the call to Switch.

The **arglist** array is an image of the target procedure's argument list as it would appear in the stack if the routine were in Compatibility Mode. Space for the function return value (if any) comes first, followed in order by the first through last parameters. Value parameters are represented by their values, and reference parameters are represented by their DB-relative addresses. Each entry in **arglist** starts on a CM 16-bit word boundary.

Note A function return value is placed in the rightmost 8 bits of the first element of the **arglist** array.

The leftmost 8 bits of **arglist(1)** is used if the NM procedure you are switching to has a parameter declared as follows:

```
TYPE bit8 = 0..255;
```

You need to use whatever construct your MPE V/E-supported programming language provides to move that 8-bit quantity into the leftmost 8 bits of the value you assign to **arglist(1)**.

The **argdesc** array provides additional information required to describe the parameters in **arglist**. Whereas **arglist** specifies the location of the parameters (value parameters are in the list; reference parameters are pointed to by the list), **argdesc** specifies the type of the parameter. There is usually a one-to-one correspondence between the entries in **argdesc** (descriptors) and the parameters to be passed. The first entry in the **argdesc** array describes the first parameter, and so on to the last entry describing the last parameter. The exception (described in the discussion of the **nparms** parameter) involves NM target procedures that have hidden parameters. You must account for any hidden parameters due to `ANYVAR` parameters, an extensible parameter list, and so forth. For more information on hidden parameters, refer to the *HP Pascal/XL Reference Manual* (31502-90002).

Note Arguments can be longer than one **arglist** element. For example, it takes two **arglist** elements to hold one 32-bit integer passed by value. The descriptor types map to parameters, not to **arglist** elements.

Figure 4-2 illustrates the relationship between the **argdesc** and **arglist** arrays:

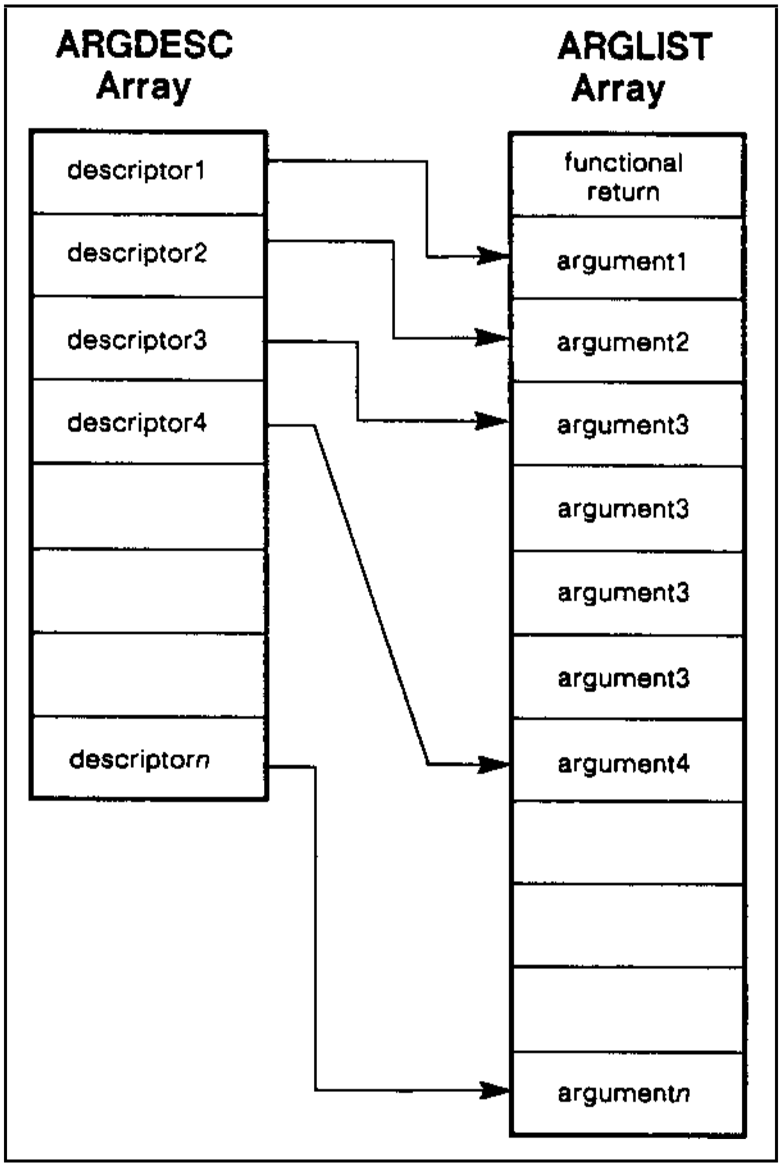


Figure 4-2. ARGDESC and ARGLIST Arrays

The descriptors in the **argdesc** array are intergers that represent the type of the associated parameter. The type IDs and their NM and CM sizes (in bits) are listed in Table 4-1. The types and their associated SPL-HP Pascal/XL, Pascal/V-HP Pascal/XL, and COBOL II/V-HP Pascal/XL mappings are listed in Tables 4-2, 4-3, and 4-4, respectively.

Types 1 through 4 direct Switch to pass by *value* the corresponding parameters you have placed in **arglist**

Types 5 and 6 are used to pass *reference* parameters only. Types 5 and 6 direct Switch to map the DB-relative addresses you place in the corresponding locations in **arglist** to equivalent MPE/XL NM virtual addresses, effectively creating NM pointers from CM pointers. The size indicated in Table 4-1 for these types is the number of bits required to contain the CM pointer to that parameter.

Types 7 through 9 should not be used. They are reserved for MPE XL.

Types 10 and 11 are used like types 5 and 6 when the target HP Pascal/XL procedure expects long pointers to reference parameters.

Types 12 and 13 are used as place holders in the target procedure's parameter list, standing for optional reference parameters that are omitted on this call. The corresponding elements(s) of the **arglist** parameter should be set to nil.

Note Nil is not necessarily 0. The Pascal standard defines nil as being implementation-dependant. You should test the value of nil in the Pascal to which you are switching if your Switch call must pass a nil pointer.

Table 4-1. Parameter Type IDs and Their NM/CM Sizes

Call Type	Type ID	NM Size (Bits)	CM Size (Bits)
By Value	1	8	8
	2	16	16
	3	32	32
	4	64	64
By Reference, Short Pointer	5	32	16
	6	32	16
	7	Do not use	Do not use
	8	Do not use	Do not use
	9	Do not use	Do not use
By reference, Long Pointer	10	64	16
	11	64	16
Optional Reference Parameters	12	32	16
	13	64	16

When calling Switch from SPL, declare two arrays as follows:

```
integer array argdesc(0:31);
integer array arglist(0:63);
```

Then, for each parameter, follow the appropriate instructions given in Table 4-2.

Table 4-2. SPL—> HP Pascal/XL Parameter Type Mappings

If the NM procedure you are switching to has a parameter declared as follows:	Declare it like this in Pascal/V:	And do the following to pass the Pascal/V variable through the switch:

`TYPE bit8 = 0..255;`
`A : bit8;`
	`byte A;`	`argdesc(n) := 1;`
		`arglist(n) := A;`

`B : shortint;`	`integer B;`	`argdesc(n) := 2;`
		`arglist(n) := B;`

`C : integer;`	`double C;`
	`integer array C'(*)`
	`= C;`
		`argdesc(n) := 3;`
		`move arglist(n) :=`
		`C'(0),(2);`

`D : longint;`
`D : anyptr;`
	`integer array`
	`D(0:3);`
		`argdesc(n) := 4;`
		`move arglist(n) :=`
		`D(0),(4);`

`TYPE pacN = packed`
`array [1..N] of`
`char;`
 	
`VAR E : pacN;`
	`byte array E(0:N-1);`	`argdesc(n) := 5;`
		`arglist(n) := @E;`

`TYPE 16_bit_ary =`
`array [1..N] of`
`shortint;`
`TYPE 32_bit_ary =`
`array [1..N] of`
`integer;`
 	
`VAR F : 16_bit_ary;` or
`VAR F : 32_bit_ary;`
	`integer array`
	`F(0:N-1);`
	`double array`
	`F(0:N-1);`
		`argdesc(n) := 6;`
		`arglist(n) := @F;`

Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.

`VAR J : anyptr;`	`byte array J(0:N);`	`argdesc(n) := 10;`
		`arglist(n) := @J;`

`K : anyptr;`	`integer array`
	`K(0:N);`
		`argdesc(n) := 11;`
		`arglist(n) := @K;`

`L : localanyptr;`	Nothing required.	`argdesc(n) := 12;`
		`arglist(n) := 0;`

`M : anyptr;`	Nothing required.	`argdesc(n) := 13;`
		`arglist(n) := 0;`

When calling Switch from Pascal/V, declare two arrays as follows:

```
TYPE
   shortint = -32768..32767;

VAR
   argdesc : array [1..32] of shortint;
   arglist : array [1..64] of shortint;
```

Then, for each parameter, follow the appropriate instructions given in Table 4-3.

Table 4-3.
Pascal/V—> HP Pascal/XL Parameter Type Mappings (Page 1)

If the NM procedure you are switching to has a parameter declared as follows:	Declare it like this in SPL:	And do the following to pass the SPL variable through the switch:

`TYPE bit8 = 0..255;`
 	
`A : bit8;`
	`TYPE bit8 = 0..255;`
	
	`A : bit8;`
		`argdesc[n] := 1;`
		`arglist[n] := A;`

`B : shortint;`	`B : shortint;`	`argdesc[n] := 2;`
		`arglist[n] := B;`

`C : integer;`	`TYPE split_word =`
	`record`
	`case integer of`
	`0 : (all : integer);`
	`1 : (hi  : shortint;`
	`lo  : shortint);`
	`end;`
	`VAR`
	`C : split_word;`
		`argdesc[n] := 3;`
		`arglist[n] := C.hi;`
		`arglist[n+1] := C.lo;`

`D : longint;` or
`D : anyptr;`
	`TYPE split_long =`
	`record`
	`case integer of`
	`0:(all : array[1..2]`
	`of shortint);`
	`1:(sh : array[1..4]`
	`of shortint);`
	`end;`
	`VAR`
	`D : split_long;`
		`argdesc[n] := 4;`
		`arglist[n] :=`
		`D.sh[1];`
		`arglist[n+1] :=`
		`D.sh[2];`
		`arglist[n+2] :=`
		`D.sh[3];`
		`arglist[n+3] :=`
		`D.sh[4];`

`TYPE pacN = packed`
`array [1..N]`
`of char;`
`VAR E : pacN;`
	`TYPE pacN = packed`
	`array [1..N]`
	`of char;`
	
	`VAR E : pacN;`
		`argdesc[n] := 5;`
		`arglist[n] :=`
		`baddress(E);`

TABLE 4.3 Ppascal/V—> HP Pascal/XL Parameter Type Mappings (Page 2)

If the NM procedure you are switching to has a parameter declared as follows:	Declare it like this in SPL:	And do the following to pass the SPL variable through the switch:

`TYPE 16_bit_ary = array [1..N] of shortint;`
`TYPE 32_bit_ary = array [1..N] of integer;`
 	
`VAR F : 16_bit_ary;` or
`VAR F : 32_bit_ary;`
	`TYPE 16_bit_ary = array [1..N] of shortint;`
	`TYPE 32_bit_ary = array [1..N] of integer;`
	
	`VAR F : 16_bit_ary;`
	or
	`VAR F : 32_bit_ary;`
		`argdesc[n] := 6;`
		`arglist[n] : waddress(F);`

Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.

`J : anyptr;`	`J : packed array [1..N] of char;`	`argdesc[n] := 10;`
		`arglist[n] := baddress(J);`

`K : anyptr;`	`K : array [1..N] of shortint;`	`argdesc[n] := 11;`
		`arglist[n] := waddress(K);`

`L : localanyptr;`	Nothing required.	`argdesc[n] := 12;`
		`arglist[n] := NIL;`

`M : anyptr;`	Nothing required.	`argdesc[n] := 13;`
		`arglist[n] := NIL;`

When calling Switch from COBOL II/V, declare two arrays as follows:

```
01 ARGDESC PIC S9(4) COMP OCCURS 32 TIMES.
01 ARGLIST PIC S9(4) COMP OCCURS 64 TIMES.
```

Then, for each parameter, follow the appropriate instructions given in Table 4-4.

Table 4-4.
COBOL II/V -> HP Pascal/XL Parameter Type Mappings(Page 1)

If the NM procedure you are switching to has a parameter declared as follows:	Declare it like this in COBOL II/V:	And do the following to pass the COBOL II/V variable through the switch:

`TYPE bit8 = 0..255;`
`A : bit8;`
	`01 A PIC X.` or
	`01 A PIC 9.` or
	`01 A PIC 9 COMP.`
		`MOVE A TO ARGLIST(N).`

`B : shortint;`	`01 B PIC S9 COMP.`
	`01 B PIC S9(2) COMP.`
	`01 B PIC S9(3) COMP.`
	`01 B PIC S9(4) COMP.`
		`MOVE B TO ARGLIST(N).`

`C : integer;`	`01 C PIC S9(5) COMP.`
	`01 C PIC S9(6) COMP.`
	`01 C PIC S9(7) COMP.`
	`01 C PIC S9(8) COMP.`
	`01 C PIC S9(9) COMP.`
	`01 C-SPLIT REDEFINES C.`
	`05 C-HI PIC S9(4) COMP.`
	`05 C-LO PIC S9(4) COMP.`
		`MOVE C-HI TO ARGLIST(N).`
		`MOVE C-LO TO ARGLIST(N+1).`

`D : longint;`	`01 D PIC S9(10) COMP.`
	`01 D PIC S9(11) COMP.`
	`01 D PIC S9(12) COMP.`
	`01 D PIC S9(13) COMP.`
	`01 D PIC S9(14) COMP.`
	`01 D PIC S9(15) COMP.`
	`01 D PIC S9(16) COMP.`
	`01 D PIC S9(17) COMP.`
	`01 D PIC S9(18) COMP.`
	`01 D-SPLIT REDEFINES D.`
	`05 D-SHORT PIC S9(4)`
	
	`COMP OCCURS 4 TIMES.`
		`MOVE D-SHORT(1) TO ARGLIST(N).`
		`MOVE D-SHORT(2) TO ARGLIST(N+1).`
		`MOVE D-SHORT(3) TO ARGLIST(N+2).`
		`MOVE D-SHORT(4) TO ARGLIST(N+3).`

`TYPE pacN = packed array[1..N] of char;`	`01 E PIC X(80).` (or any size array)	`CALL ".LOC." USING E GIVING ARGLIST(N).`

Table 4-4. COBOL II/V -> HP Pascal/XL Parameter Type Mappings(Page 2)

If the NM procedure you are switching to has a parameter declared as follows:	Declare it like this in COBOL II/V:	And do the following to pass the COBOL II/V variable through the switch:
Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.
Reserved for MPE XL.	Do not use.	Do not use.
`VAR J : anyptr;`	`01 J PIC X(80) OCCURS N TIMES.`	`CALL ".LOC." USING J GIVING ARGLIST(N).`
`K : anyptr;`	`01 K PIC S9(9) COMP OCCURS N TIMES.`	`CALL ".LOC." USING K GIVING ARGLIST(N).`
`L : localanyptr;`	Nothing required.	`MOVE 0 TO ARGLIST(N).`
`M : anyptr;`	Nothing required.	`MOVE 0 TO ARGLIST(N).`

Function returns are described using a similar scheme. Those mappings are given in Figure 4-3.

TYPE ID	SPL Function Type	CM Size (Bytes)	Pascal/XL Function Type	NM Size (Bytes)
0	none	(0)	not a function	(0)
1	byte	(1)	bit8	(1)
2	integer	(2)	shortint	(2)
3	double	(4)	integer	(4)
4	long	(8)	longreal	(8)

Figure 4-3. Supported Function Type Mappings

Note

A function return value is placed in the rightmost 8 bits of the first element of the **arglist** array. The leftmost 8 bits of **arglist(1)** is used if the NM procedure you are switching to has a parameter declared as follows:

```
TYPE bit8 = 0..255;
```

You need to use whatever construct your MPE V/E-supported programming language provides to move that 8-bit quantity into the leftmost 8 bits of the value you assign to **arglist(1)**.

HPSWTONM PLABEL Intrinsic

The `HPSWTONMPLABEL` intrinsic allows CM user programs, user libraries, and system code to invoke NM procedures. In a manner similar to the `HPSWITCHTOCM` intrinsic, `HPSWTONMPLABEL` does the following:

- Converts the CM references in the argument list to virtual NM addresses
- Changes the execution mode
- Invokes the NM procedure specified by the CM caller

Since NM code can address the entire CM stack, there is no copying of reference parameters. The length of each actual parameter is not needed because lengths are implied in the descriptor list.

The syntax of this intrinsic and detailed explanations of its parameters are given in the following paragraphs. Also provided are examples of switches in the CM—> NM direction.

Syntax

Prior to calling the `HPSWTONMPLABEL` intrinsic, your programming language may require you to declare it. In Pascal/V, the declaration is as follows:

```
        FUNCTION HPSWTONMPLABEL : INTEGER; INTRINSIC;
```

Next, an example Pascal/V call to this intrinsic:

```
        return_status := HPSWTONMPLABEL (proc, nparms, arglist,
                                    argdesc, functype);
```

You call the `HPSWTONMPLABEL` intrinsic with five parameters. These parameters provide Switch with the following information:

- Plabel of the NM routine (as returned by the `HPLOADNMPROC` intrinsic)
- Number of parameters in the NM routine
- Parameter list
- Parameter description list
- Type of the functional return value (if any)

All this information is key to the correct operation of mixed-mode procedure calls.

Parameters

A detailed explanation of the parameters of the `HPSWTONMPLABEL` intrinsic follows.

Required parameters are shown in **boldface**; optional parameters are shown in *italics*.

proc **double by value (required)**

Passes the NM plabel of the target procedure name. This plabel is usually obtained by calling the `HPLOADNMPROC` intrinsic.

nparms **16-bit signed integer by value (required)**

Passes the number of parameters you are passing to the target NM procedure. It specifies the length of the **argdesc**\array. You must also account for any hidden parameters due to `ANYVAR` parameters, an extensible parameter list, and so forth. For more information on hidden parameters, refer to the *HP Pascal/XL Reference Manual* (31502-90002).

arglist **16-bit signed integer array by reference (required)**

Passes the actual parameters you are going to pass to the NM procedure.

argdesc **16-bit signed integer array by reference (required)**

Passes integer codes describing the parameters held in the **arglist** array, that is, byte, word, double, pointer, and so forth.

functype **16-bit signed integer by value (required)**

Passes the data type of the value the target procedure returns if it is a function. If the target is not a function, the value of this parameter is zero. The supported function types are described in Table 4-5.

For more information on the **arglist** and **argdesc** arrays, see the discussion of `HPSWTONMNAME` parameters.

HPLOADNM PROC Intrinsic

The `HPLOADNMPROC` intrinsic returns the plabel of an NM procedure.

You call this intrinsic to obtain the NM plabel of the target procedure. This plabel is then used by the `HPSWTONMPLABEL` intrinsic as the value of its **proc** parameter.

Syntax

Prior to calling `HPLOADNMPROC`, your programming language may require you to declare it. In Pascal/V, the declaration is as follows:

```
        FUNCTION HPLOADNMPROC : INTEGER; INTRINSIC;
```

Next, an example of a Pascal/V call to `HPLOADNMPROC`:

```
        plabel := HPLOADNMPROC (procname, proclen, libname,
                                liblen);
```

You call the `HPLOADNMPROC` with four parameters. These parameters supply the following information:

- Name of the NM target procedure
- Length of the name of the NM target procedure
- Name of the library to be searched for the NM target procedure
- Length of the name of the library to be searched

Caution The strings you supply as values of the **procname** and **libname** parameters must exactly match the names of the target NM routine and its NM library, respectively.

Parameters

A detailed explanation of the parameters of the `HPLOADNMPROC` intrinsic follows.

Required parameters are shown in **boldface**; optional parameters are shown in *italics*.

procname	**byte array by reference (required)** Passes the target procedure name. The target procedure must be contained in an Executable Library. If the value of **procname** is invalid, blank, or does not contain your NM procedure, `NL.PUB.SYS` is searched.
proclen	**16-bit signed integer by value (required)** Passes the byte length of the procedure name.
libnamebyte array by reference (required) Passes the name of the NM library to be searched for the target. If the value of **libname**	is invalid, blank or does not contain your NM procedure, `NL.PUB.SYS` is searched.
liblen	**16-bit signed integer by value (required)** Passes the byte length of the library name.

Examples: CM to NM and Return

Now consider an example of the mixed-mode switching process in the CM—
> NM direction. The `HPCIDELETEVAR` intrinsic removes an entry from
the session-local variable table. This intrinsic is not directly callable from
Compatibility Mode. However, you can call it from CM code by means of a
CM—> NM Switch stub.

The syntax of the `HPCIDELETEVAR` intrinsic is as follows:

```
                     CA      I32
      HPCIDELETEVAR(varname, status);
```

The **varname** parameter is a required character array. It passes the name of the
variable to be deleted. The name can be up to 255 characters in length and
must be a valid MPE XL variable name.

The *status* parameter is an optional 32-bit signed integer passed by reference.
It returns a number indicating the status of the procedure. The default is nil.

For more information on `HPCIDELETEVAR`, refer to the appropriate entry in the
MPE XL Intrinsics Reference Manual (32650-90028).

Figure 4-4 illustrates the purpose of the CMDeleteVar Switch stub.

Figure F04-03 here.

Figure 4-4. HPSWTONMNAME Example, CMDeleteVar

The Switch stub sets up the parameters required by the appropriate Switch intrinsic. In this instance, that is the `HPSWTONMNAME` intrinsic. Here, again, is a sample call to `HPSWTONMNAME`:

```
return_status := HPSWTONMNAME (procname, proclen, libname,
                      liblen, nparms, arglist, argdesc,
                      functype);
```

The parameters that the CMDeleteVar Switch stub must set up before it can call the `HPSWTONMNAME` intrinsic convey to Switch the following information:

- Name of the NM routine
- Length of the procedure name
- NM library to search for the target procedure
- Length of the library name
- Number of parameters of the NM routine
- Parameter list
- Parameter description list
- Type of the functional return value (if any)

Example 4-1 contains the complete Switch to NM stub.

Example 4-1. CMDeleteVar Stub

```
{ XAMPL41 -- Switch to NM by name }
$standard_level 'HP3000'$
{$subprogram$} {uncomment this to make an RBM for your SL}
$uslinit$

PROGRAM XAMPL41(input, output);

{Type Declarations}

TYPE
   shortint = -32768..32767;
   shr_ary32 = packed array [1..32] of shortint;
   xlstatus = record
      case integer of
         0 : (all : integer);
         1 : (info    : shortint;
              subsys  : shortint);
      end;
   pac16 = packed array [1..16] of char;
   pac255 = packed array [1..255] of char;

{Global variable declarations}

VAR

{Parameters passed to HPCIDELETEVAR via Switch}

   status : xlstatus; {status must be 4-byte aligned}
                {waddress must return an even number}
   VarName : pac255;

{Intrinsic procedure declarations}

FUNCTION HPSWTONMNAME : integer; intrinsic;

FUNCTION HPSWTONMPLABEL : integer; intrinsic;
FUNCTION HPLOADNMPROC : integer; intrinsic;
```

Example 4-1. CMDeleteVar Stub, continued

```
{Stub procedure declaration}

PROCEDURE CMDeleteVar(VAR CIVarName : pac255;
                      VAR NMStatus  : xlstatus);

VAR

{Switch intrinsic parameters}

   arglist : shr_ary32;  {parameter list}
   argdesc : shr_ary32;  {parameter description list}
   fct_typ : shortint;   {functional return type, if any}
   lib_name : pac16;     {NM library to search for target}
   lib_len : shortint;   {length of NM library name}

   nparms : shortint;    {number of target parameters}
   proc_name : pac16;    {name of NM routine}
   proc_len : shortint;  {length of target's name}

{Parameter assigned functional return}

   rtn_st : integer;

BEGIN  {stub procedure CMDeleteVar}

{Initializations}

   proc_name := 'HPCIDELETEVAR   ';
   proc_len  := 13;
   lib_name  := 'NL.PUB.SYS      ';
   lib_len   := 10;

   nparms    := 3;    {nparms governs how many types are}
                      {searched for in argdesc;         }
                      {2 parms plus extensible_gateway  }
                      {HPCIDELETEVAR is declared        }
                      {$OPTION 'EXTENSIBLE_GATEWAY$     }
```

```
arglist[1] := 0;  {extensible gateway mask is 32 bits}
arglist[2] := 0;  {of anything, all 0 bits works ok! }

arglist[3] := baddress(CIVarName);
           {reference parameter passed by address;   }
           {take byte address of variable name buffer}

arglist[4] := waddress(NMStatus);
                {reference parameter passed by address;}
                {take word address of local status}
```

Example 4-1. CMDeleteVar Stub, continued

```
      argdesc[1] := 03; {32-bit word value for gateway mask}
      argdesc[2] := 05; {byte pointer for arglist[3]}
      argdesc[3] := 06; {word pointer for arglist[4]}

      fct_typ := 00; {This is an NM procedure, not a       }
                     {function.  If it was an NM function, the }
                     {return value would come back in         }
                     {arglist[1..n] where n is the length of   }
                     {the value in 16-bit words.               }

   {Switch intrinsic call}

      rtn_st := HPSWTONMNAME (proc_name,
                              proc_len,
                              lib_name,
                              lib_len,
                              nparms,
                              arglist,
                              argdesc,
                              fct_typ);

   {Since Status was passed by reference, if the     }
   {Switch succeeded, HPCIDELETEVAR will have set it.}
   {Otherwise, the Switch failed, so return the      }
   {status from the Switch.  This works because      }
   {.subsys is unique.                               }

      if (rtn_st <> 0) then
         NMStatus.all := rtn_st;

   END;  {Stub procedure CMDeleteVar}

   BEGIN  {outer block}

      writeln('Use the command "SETVAR DELETE_ME 0" before');
      writeln('running this.');
      writeln(' ');
   VarName := 'DELETE_ME';
   status.all := 0;

   CMDeleteVar(VarName, status);

   writeln('Status for subsystem ', status.subsys:3);
   writeln(' is ', status.info:3);

END.  {outer block}

{end Example 4-1}
```

Note For a complete analysis of CM—> NM Switch stub code, refer to Chapter 5.

Figure 4-5 illustrates how your CMDeleteVar Switch stub accesses the `HPCIDELETEVAR` intrinsic in `NL.PUB.SYS`.

Figure F04-04 here.

Figure 4-5. CM—> NM Switch Summary 2

Example 4-2 illustrates a CM—> NM Switch, using a COBOL II/V stub procedure in Compatibility Mode to access an HP Pascal/XL target procedure in Native Mode. Both the stub and the target are included in Example 4-2.

Example 4-2. CM—> NM Switch, COBOL

```
{ XAMPL42 -- example Switch to NM using COBOL stub}
$CONTROL USLINIT
 IDENTIFICATION DIVISION.
 PROGRAM-ID. XAMPL42.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  SWITCH-STATUS             PIC S9(9) COMP VALUE ZERO.
 01  SWITCH-STATUS-X REDEFINES SWITCH-STATUS.
     05 SWITCH-INFO            PIC S9(4) COMP.
     05 SWITCH-SUBSYS          PIC S9(4) COMP.
 01  DISP-INFO                 PIC ZZZ9.
 01  DISP-SUBSYS               PIC ZZZ9.
 01  DISP-ANSWER               PIC ZZZ9.
 01  PROCNAME                  PIC X(16) VALUE SPACES.
 01  ADD-TO-PARM               PIC S9(9) COMP.
 01  PROCNAME-LEN              PIC S9(4) COMP.
 01  LIBNAME                   PIC X(80) VALUE SPACES.
 01  LIBNAME-LEN               PIC S9(4) COMP.
 01  ARGLIST.
     05 LIST-ELEMENT OCCURS 32 TIMES PIC S9(4) COMP.
 01  ARGDESC.
     05 DESC-ELEMENT OCCURS 32 TIMES PIC S9(4) COMP.
 01  NPARMS                    PIC S9(4) COMP.

PROCEDURE DIVISION.
BEGIN.

     DISPLAY "Begin execution of CM main PROG".

*
* SET THE "REFERENCE" PARAMETER TO HAVE A VALUE OF 1
```

```
*

      MOVE 1 TO ADD-TO-PARM.
```

Example 4-2. CM—> NM Switch, COBOL, continued

```
*
* ESTABLISH PROCNAME, LIBNAME, AND ASSOCIATED LENGTHS.
*

      MOVE "testadd" TO PROCNAME.
      MOVE 7 TO PROCNAME-LEN.
      MOVE "NL" TO LIBNAME.
      MOVE 2 TO LIBNAME-LEN.
      MOVE 2 TO NPARMS.

*
* BUILD THE ARGUMENT LIST ARRAY:
*   1) 0 RESERVES SPACE FOR THE RETURN VALUE
*   2) 99 RESERVES SPACE FOR THE FIRST PARAMETER (BY VALUE),
* CALL .LOC. INTRINSIC TO GET ADDRESS OF THE
*   BY-REFERENCE PARAMETER
*

      MOVE 0 TO LIST-ELEMENT( 1 ).
      MOVE 99 TO LIST-ELEMENT( 2 ).
      CALL INTRINSIC ".LOC." USING ADD-TO-PARM
                             GIVING LIST-ELEMENT( 3 ).

*
* BUILD THE ARGUMENT DESCRIPTOR ARRAY
*

      MOVE 2 TO DESC-ELEMENT( 1 ).
      MOVE 6 TO DESC-ELEMENT( 2 ).

*
* MAKE THE SWITCH CALL
*
```

```
     CALL "HPSWTONMNAME" USING @PROCNAME, \PROCNAME-LEN\,
                               @LIBNAME, \LIBNAME-LEN\,
                               \NPARMS\,
                               ARGLIST, ARGDESC,
                               \2\
                         GIVING SWITCH-STATUS.

*
* TEST STATUS
*

     IF SWITCH-STATUS IS NOT ZERO
        MOVE SWITCH-INFO TO DISP-INFO
        MOVE SWITCH-SUBSYS TO DISP-SUBSYS
        DISPLAY "Status info = ", DISP-INFO
        DISPLAY "Status subsys = ", DISP-SUBSYS
     ELSE
        DISPLAY "HPSWTONMNAME completed successfully".
```

Example 4-2. CM—> NM Switch, COBOL, continued

```
*
* SHOW THE RETURN VALUE (WHICH SHOULD BE 100)
*

     MOVE LIST-ELEMENT( 1 ) TO DISP-ANSWER.
     DISPLAY "Return value = ", DISP-ANSWER.

     STOP RUN.

*
* END COBOL II/V PROGRAM IN COMPATIBILITY MODE
*

{ Target Pascal/XL procedure in Native Mode }

$subprogram$
```

```
program dummy_outer_block(input, output);

TYPE
    A8_INTEGER    =    $ALIGNMENT 1$ INTEGER;

FUNCTION TESTADD(
                 byvalueparm  : SHORTINT;
             VAR byrefparm    : A8_INTEGER
                ) : SHORTINT;

BEGIN {function testadd }

   TESTADD := byvalueparm + byrefparm;

END; {function testadd }

BEGIN {dummy_outer_block }

END. {dummy_outer_block }

{end Example 4-2 }
```

Switch to NM Sequence

Examples 4-3 through 4-8 are a series of routines, intended to illustrate the partial recompilation migration level, using a CM—> NM switch.

In these examples the target procedure migrates to Native Mode while the calling routine remains in Compatibility Mode.

A possible reason for moving the target procedure to Native Mode is that it has a feature that is no longer available on MPE XL and an attractive alternative exists in the NM environment. Example 4-3 is a Pascal/V source with a procedure call to the PTAPE intrinsic (deleted on MPE XL).

Example 4-3. Pascal/V Program Calling Routine With Deleted Featur

```
PROGRAM XAMPL43(input,output);

TYPE
   pac37    = packed array [1..37] of char;
   logical  = 0..65535;
   shortint = -32768..32767;

VAR
   fname1     : pac37;
   fname2     : pac37;
   filenum1   : shortint;
   filenum2   : shortint;
   foptions1  : logical;
   foptions2  : logical;
   aoptions1  : logical;
   aoptions2  : logical;
   recsize1   : shortint;
   recsize2   : shortint;
   dev1       : shortint;
   dev2       : shortint;

function fopen : shortint; intrinsic;
procedure ptape (fnum1, fnum2 : shortint); external;

BEGIN
   fname1 := '$STDIN ';
   foptions1 := 0;
   aoptions1 := 0;
   recsize1 := 0;
   dev1 := 0;
   filenum1 := FOPEN(fname1, foptions1, aoptions1,
                     recsize1, dev1);
```

```
writeln('fopen ',fname,'ccode is ',ccode);

fname2 := 'XMPLA ';
foptions2 := 0;
aoptions2 := 0;
recsize2 := 0;
dev2 :=0;
filenum2 := FOPEN(fname2, foptions2, aoptions2,
                  recsize2, dev2);
writeln('fopen ',fname2,'ccode is ',ccode);
```

Example 4-3. Pascal/V Program, continued

```
      PTAPE(filenum1,filenum2);
      writeln('ptape result: ');
      case ccode of
         0 : writeln('CCE');
         1 : writeln('CCG');
         2 : writeln('CCL');
      end;
   END.

   {end Example 4-3}
```

Example 4-4 is an SPL procedure incorporating the call to the `PTAPE` intrinsic.

Example 4-4. SPL Procedure With Feature Deleted on MPE XL

```
   $CONTROL USLINIT,SUBPROGRAM,SEGMENT=XAMPL43
   BEGIN
      EQUATE BASE = 10;
      DEFINE D'MININT = -214748256D#;
      EQUATE OFFSET = 48;
      EQUATE PLACES = 9;    << size of maxint, 32-bits >>

   INTRINSIC FREAD, FWRITE;

   PROCEDURE PTAPE(FILE1, FILE2);
      VALUE FILE1, FILE2;
      INTEGER FILE1, FILE2;
   << PTAPE replaces the deleted MPE/V intrinsic of the    >>
   << same name.  This routine transfers records from      >>
   << FILE1 to FILE2.                                      >>
   BEGIN
      INTEGER RECLEN, LEN;
      LOGICAL ARRAY BUFFER(1:256);
      RECLEN := 256;
      DO
         BEGIN
            LEN := FREAD(FILE1, BUFFER, RECLEN);
            IF = THEN FWRITE(FILE2, BUFFER, LEN, 0);
      END
   UNTIL <>;
END;

END.

{end Example 4-4}
```

To replace the deleted feature, you may want to write your own NM procedure. Example 4-5 is an HP Pascal/XL procedure replacing the call to the deleted PTAPE intrinsic with an MPE XL feature. This procedure will replace the SPL procedure.

Example 4-5. Pascal/XL Procedure Replacing Deleted Feature

```
$standard_level 'ext_modcal'$
$subprogram$

PROGRAM XAMPL45 (input,output);

TYPE
   pac256 = packed array [1..256] of char;

FUNCTION FREAD : SHORTINT; INTRINSIC;
PROCEDURE FWRITE; INTRINSIC;

PROCEDURE PTAPE(FILE1, FILE2 : integer);

{ PTAPE replaces the deleted MPE/V intrinsic of the  }
{ same name.  This routine transfers records from    }
{ FILE1 to FILE2.                                    }

VAR
   reclen : shortint;
   len : shortint;
   buffer : pac256;

BEGIN
   reclen := 256;
   repeat
      begin
         len := FREAD(file1, buffer, reclen);
         if ccode = 2 then FWRITE(file2, buffer, len, 0);
      end
   until (ccode <> 2);
END;
```

```
begin
end.

{end Example 4-5}
```

It is advisable to test one thing at a time. You can test the NM replacement procedure separately from a Switch stub or in-line Switch that calls the appropriate intrinsic to invoke the target. Example 4-6 is an NM driver to test the HP Pascal/XL procedure.

Example 4-6. NM Driver

```
$standard_level 'ext_modcal'$

PROGRAM XAMPL46(input,output);

TYPE
   pac37    = packed array [1..37] of char;
   logical  = 0..65535;

VAR
   fname1     : pac37;
   fname2     : pac37;
   filenum1   : shortint;
   filenum2   : shortint;
   foptions1  : logical;
   foptions2  : logical;
   aoptions1  : logical;
   aoptions2  : logical;
   recsize1   : shortint;
   recsize2   : shortint;
   dev1       : shortint;
   dev2       : shortint;

FUNCTION FOPEN : SHORTINT; INTRINSIC;

PROCEDURE ptape (filenum1, filenum2 : shortint); external;

BEGIN
   fname1 := '$STDIN ';
   foptions1 := 0;
   aoptions1 := 0;
   recsize1 := 0;
   dev1 := 0;
```

```
filenum1 := FOPEN(fname1, foptions1, aoptions1,
                  recsize1, dev1);
writeln('fopen ',fname1,'ccode is ',ccode);

fname2 := 'PTAPE ';
foptions2 := 0;
aoptions2 := 0;
recsize2 := 0;
dev2 := 0;
filenum2 := FOPEN(fname2, foptions2, aoptions2,
                  recsize2, dev2);
writeln('fopen ',fname2,'ccode is ',ccode);
```

Example 4-6. NM Driver, continued

```
      ptape(filenum1,filenum2);
      writeln('ptape result: ');
      case ccode of
         0 : writeln('CCE');
         1 : writeln('CCG');
         2 : writeln('CCL');
      end;
   END.

   {end Example 4-6}
```

As it stands, the HP Pascal/XL procedure represents an unresolvable external reference to the CM routine that must call it. Again, you can use either a Switch stub or an in-line Switch to enable the CM caller to access the NM target. Example 4-7 is a Pascal/V Switch stub enabling access to the HP Pascal/XL procedure.

Example 4-7. Pascal/V Switch Stub

```
   $standard_level 'HP3000'$

   PROGRAM XAMPL47(input, output);

   TYPE
      shortint = -32768..32767;

   FUNCTION HPSWTONMNAME : integer; intrinsic;

   PROCEDURE ptape(file1, file2 : shortint);

   TYPE
      pac256 = packed array [1..256] of char;
      pac16 = packed array [1..16] of char;
      shr_ary32 = packed array [1..32] of shortint;

   VAR
      proc_name   : pac16;
      proc_len    : shortint;
```

```
lib_name    : pac16;
lib_len     : shortint;
nparms      : shortint;
arglist     : shr_ary32;
argdesc     : shr_ary32;
fct_type    : shortint;
rtn_st      : integer;
```

Example 4-7. Pascal/V Switch Stub, continued

```
BEGIN
   proc_name   := 'PTAPE           ';
   proc_len    := 5;
   lib_name    := 'SL              ';
   lib_len     := 2;

   nparms      := 2;

   arglist[1]  := file1;
   arglist[2]  := file2;

   argdesc[1]  := 02;          { 16-bit value }
   argdesc[2]  := 02;          { 16-bit value }
   fct_type     := 0;           {not a function }

   rtn_st := HPSWTONMNAME(proc_name, proc_len, lib_name,
                          lib_len, nparms, arglist,
                          argdesc, fct_type);
END;

BEGIN
END.

{end Example 4-7}
```

To compile the Switch stub in Example 4-7, use the `:PASCAL` command. For complete information, refer to the *MPE XL Commands Reference Manual* (32650-90003).

On the CM side, you must use the MPE Segmenter to put the stub into the CM SL in place of the SPL target procedure. Example 4-8 illustrates an MPE Segmenter session to replace the SPL procedure with the Switch stub in Example 4-7.

Example 4-8. Segmenter Session

```
:SEGMENTER
HP32050A.02.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO 1985
-SL SL
-LISTSL

SL FILE SL.SWITCH.EXAMPLE

SEGMENT   0 SWCMLIB         LENGTH  706

   ENTRY POINTS    CHECK CAL STT ADR
   PTAPE             3    C    1   13

   EXTERNALS       CHECK STT SEG
   FWRITE            0    13    ?
   FREAD             0    12    ?

USED     4600(23.0)  AVAILABLE 2334200(11565.0)

-PURGESL PTAPE
-USL $OLDPASS
-ADDSL PTAPE
-EXIT

END OF SUBSYSTEM

:
```

Special Considerations and Restrictions

There are certain considerations to bear in mind when you use Switch.

General

General considerations apply no matter what the direction of the mixed-mode call:

- Mixed-mode procedure calls require an increased level of programming complexity.
- The overhead of Switch can be significant.
- Since Switch does no parameter type or alignment checking, the target procedure must make the necessary checks.

The following considerations apply to a particular direction of mixed-mode call.

CM—> NM

When making mixed-mode procedure calls in the direction CM—> NM, be mindful of the following:

- Any NM procedure called from CM code must reside in an Executable Library. The CM program and its user must have access to the NL, and the NL must be properly described to `HPLOADNMPROC` or `HPSWTONMNAME`. All library searches default to `NL.PUB.SYS`.
- An NM procedure that is declared `Option Extensible_Gateway` must have a 32-bit zero placed in the first and second elements of the argument list. The descriptor list should hold a 3 (type 3, 32 bits) at the first element. The actual arguments of the procedure follow.

Testing and Debugging Considerations

You must test Switch stubs and inline Switches in the run-time environment on an MPE XL-based system. You must either compile the stub and install the compiled version in the appropriate system library or merge it with the code requiring the Switch call and recompile that merged source. You can also compile the stub and then link it with the USL/SOM file for the program that calls it. Before you can verify correct execution of the stub, the target procedure that Switch is to invoke must be present in the appropriate callable library of the other mode. You can use the MPE XL debugging facilities to troubleshoot, should you encounter problems.

The following are testing and debugging considerations that apply specifically to switches in the CM—> NM direction:

- Verify that the other-mode procedure operates correctly by calling it from its own mode.
- Verify that the switch worked correctly by checking its *status* parameter return. It should be all zeros.
- Verify that the parameter types in the **argdesc** description are correct.
- Verify that **arglist** elements were correctly reserved for hidden parameters and/or function return values.
- Verify that name lengths are correct.
- Verify that `HPLOADNMPROC` succeeded (by testing the condition code) before calling `HPSWTONMPLABEL`.
- Verify either that alignments match or that the target can handle any alignment.

5

Writing Switch Stubs

You will find the information in this and the following chapter of value if you fall into one of these categories:

- You must modify the stubs generated by the Switch Assist Tool because of the special nature of your application (see "Special Cases" in Chapter 2).
- You choose to write your own Switch stubs.
- You are the inquisitive type whose curiosity compels you to take things apart and find out how they work.

Included in this chapter are the following:

- Step-by-step analysis of writing Switch stubs
- Exercises to check comprehension and progress
- Summary checklists to follow on your own

The examples used in this chapter approximate those generated by the Switch Assist Tool in degree of complexity.

Switching to CM

To illustrate a mixed-mode switch to Compatibility Mode (CM), consider the example of an SPL procedure DECMADD that adds two packed decimal numbers, `OPERAND1` and `OPERAND2`, and leaves a packed decimal result in `RESULT`. `DECMADD` also has a parameter `DIGITS` that contains the total number of whole digits in each parameter and a parameter `FRAC` that contains the total number of fractional digits in each parameter.

The SPL declaration portion of the DECMADD procedure follows:

```
    +---------+
+----| MPE V/E |--------------------------------------+
|    +---------+                                      |
|                                                     |
|  PROCEDURE DECMADD(OPERAND1, OPERAND2, RESULT,       |
|                          DIGITS, FRAC);             |
|            VALUE         DIGITS, FRAC;              |
|            BYTE ARRAY    OPERAND1, OPERAND2, RESULT;  |
|            INTEGER       DIGITS, FRAC;              |
|                                                     |
|                                                     |
+-----------------------------------------------------+
```

`OPERAND1`, `OPERAND2`, `DIGITS`, and `FRAC` are input parameters, while `RESULT` is an output parameter.

`DIGITS` and `FRAC` are declared as value parameters. `OPERAND1`, `OPERAND2`, and `RESULT`, on the other hand, are reference parameters.

To access the DECMADD procedure from Native Mode (NM) code, you need to write a Switch stub and place that stub procedure in an Executable Library. This stub must call the appropriate Switch intrinsic (`HPSWITCHTOCM`), which, in turn, calls DECMADD.

An example call to this procedure would be as follows:

```
DECMADD(x,y,z,3,2)
```

where `x` = 100.01 and `y` = 156.86. Upon return from the procedure, `z` = 256.87.

Writing the DECMADD Stub Declarations

Begin the process of writing a Switch stub with the declaration portion of the procedure. To guarantee that the switching process is transparent to the calling program, follow these guidelines:

- Make the stub name identical to that of the SPL procedure.
- Make the types of the stub parameters correspond to those of the SPL procedure.

Type correspondence refers both to status as either value or reference parameter, as well as to the data type of the parameter. In SPL, value parameters are explicitly declared. In HP Pascal/XL, it is reference parameters that are explicitly declared by being preceded by the designation `VAR`. Data type correspondence is not quite so easily obtained. To match the SPL `INTEGER` and `BYTE ARRAY` data types in HP Pascal/XL, you need to use declared types.

The declaration portion includes the following:

- Stub header
- Stub parameters
- Called external procedures

Declaring the Stub Header

The following extract is a possible declaration portion for the stub procedure to enable a call to the SPL DECMADD procedure:

```
     +--------+
+----| MPE XL |-------------------------------------------+
|    +--------+                                           |
|                                                         |
| TYPE                                                    |
|    decm_number : PACKED ARRAY [1..80] OF CHAR;          |
|                                                         |
| PROCEDURE DECMADD(VAR OPERAND1, OPERAND2 : decm_number; |
|                   VAR RESULT : decm_number;             |
|                   DIGITS, FRAC  :  shortint);           |
|                                                         |
+-----------------------------------------------------------+
```

Note

The data type `shortint` is used to correspond to the SPL `INTEGER` type because DECMADD is a CM routine and integers in Compatibility Mode are 16-bit.

Declaring the Stub Parameters

The next step in writing a Switch stub is to set up the parameters required by the particular Switch intrinsic, in this instance `HPSWITCHTOCM`.

Consider again the declaration of the `HPSWITCHTOCM` intrinsic:

```
PROCEDURE HPSWITCHTOCM; INTRINSIC;
```

Next comes an example of an HP Pascal/XL call to `HPSWITCHTOCM`:

```
HPSWITCHTOCM(proc, method, numparms, parms,
      funcreturnlen, funcvalue, conditioncode,
      userstatus);
```

You call the `HPSWITCHTOCM` intrinsic with eight parameters. These parameters provide Switch with the following information:

- Name and CM library or the plabel of the target procedure.
- Whether the target procedure runs in normal, split-stack, or no-copy mode.
- Number of parameters being passed to the target procedure.
- A list containing a description of each parameter being passed, including:
  - Pointer to each parameter; that is, a reference to where the parameter begins in NM memory (value parameters larger than one byte must be 16-bit aligned).
  - Length (size) of each parameter in bytes (must be positive integer <= 2 ** 16).
  - For reference parameters, the stub must specify whether it is a byte or word address and also indicate whether the parameter is an input and/or output parameter.
- Length of the function return value (0 if not a function).
- Pointer to the function return value (nil if not a function).
- CM condition code value for the target procedure.
- Status record to report on `HPSWITCHTOCM`'s operation.

Setting up these parameters involves three stages:

1. Declaring the necessary constants
2. Declaring the necessary user-defined types
3. Declaring the necessary variables

Declaring Constants. Constants are declared to correspond to the following possibilities:

- How does the operating system find the target procedure?
  - By number
  - By name
  - By plabel
- Which CM library is the target procedure in?
  - System SL
  - Logon public SL
  - Logon group SL
  - Program's public SL
  - Program's group SL
- Is the target procedure called in split-stack mode?
- Of what type is a parameter?
  - Value parameter
  - Reference parameter requiring a word address
  - Reference parameter requiring a byte address
- What are the valid condition and status codes?

The excerpt in Example 5-1 is a typical constant declaration section for a Switch stub procedure:

Example 5-1. DECMADD Stub, Constant Declarations

```
const

{The OS finds procedure either by number, by name, or  }
{by plabel.                                            }

    pidt_known = 0;          {it is found by number   }
    pidt_load  = 1;          {it must be found by name}
    pidt_plabel = 2;         {it is found by plabel   }

{Which library is the procedure in?}

    system_sl       = 0;
    logon_pub_sl    = 1;
    logon_group_sl  = 2;
    pub_sl          = 3;
    group_sl        = 4;

{Is the procedure split-stack callable?}

    method_normal = 0;           {non-split-stack     }
    method_split  = 1;           {split-stack callable}
    method_no_copy =2;           {no-copy method      }

{What is the parameter type?}

    parm_type_value     = 0;  {value parameter         }
    parm_type_word_ref  = 1;  {word address is required}
    parm_type_byte_ref  = 2;  {byte address is required}

{Condition and status code constants}

    ccg    = 0;             {condition code greater (>)}
    ccl    = 1;             {condition code less (<)   }
    cce    = 2;             {condition code equal (=)  }
    All_Ok = 0;             {Used in status check      }

{end Example 5-1}
```

Declaring User-Defined Types. Several user-declared types are necessary to set up the stub procedure variables. These include the following:

- Type to represent parameter names (a packed array of characters)
- Variant type to represent the `HPSWITCHTOCM` **proc** parms parameter (contents depend on whether the target procedure is called by number, by name, or by plabel)
- Array of records type to represent the `HPSWITCHTOCM` **parms** parameter, where the individual records are declared to be a packed record type having the following components:
  - Pointer to where the parameter is located
  - Length value indicating the size of the parameter
  - Value indicating parameter type (value, word-address reference, or byte-address reference)
  - For reference parameters, a value indicating status as either an input, output, or input/output parameter
- Condition and status types

The type declaration section for the DECMADD stub procedure follows in Example 5-2:

Example 5-2. DECMADD Stub, Type Declarations

```
type

    bit8                = 0..255;
    bit16               = 0..65535;
    bit8_a1             = $ALIGNMENT 1$ bit8;
    bit16_a1            = $ALIGNMENT 1$ bit16;

{type declaration for procedure names}

    cm_proc_name = packed array [1..16] of char;

{defining generic buffer type}

    generic_buffer = packed array [1..65535] of char;

{defining type of HPSWITCHTOCM proc parameter}

    scm_procedure  =   packed record   {variant record}
        case p_proc_id_type : bit8 of

   {proc found by number}
            pidt_known : ( p_fill    : bit8_a1;
                           p_proc_id : bit16_a1 );

   {proc found by name}
            pidt_load : ( p_lib       : bit8_a1;
                          p_proc_name : cm_proc_name);

   {proc found by plabel}
            pidt_plabel : ( p_plabel : bit16_a1);
         end;  {record}

{defining type of indicator as input and/or output       }
{parameter                                               }

    scm_io_type   =   set of ( INPUT_PARM, OUTPUT_PARM );
```

```
{define individual record of HPSWITCHTOCM parms parameter;}
{parms is array describing the stub parameters;          }
{each record describes a parameter                       }

    parm_desc   =   packed record
        pd_parmptr : globalanyptr;  {where parameter found}
        pd_parmlen :   bit16;       {size in bytes        }
        pd_parm_type : bit16;       {byte or word address }
        pd_io_type :   scm_io_type; {input and/or output  }
     end;

{defining type of HPSWITCHTOCM parms parameter}

    scm_parm_desc_array = array [0..31] of parm_desc;
```

Example 5-2. DECMADD Stub, Type Declarations, Continued

```
{defining condition code type}

    ccode_type = shortint;

{defining status code type}

    xlstatus =
       record
          case integer of
             0 : (all : integer);
             1 : (info : shortint;
                  subsys : shortint);
          end; {record}

{end Example 5-2}
```

Declaring Variables. You declare two groups of variables in the variable declaration section of the stub:

- Eight variables required by the HPSWITCHTOCM intrinsic
- Any local variables

The variable declaration section for the DECMADD stub follows in Example 5-3:

Example 5-3. DECMADD Stub, Variable Declarations

```
var

    proc       : scm_procedure;     {target procedure name}
    parms      : scm_parm_desc_array;   {describes target }
                                        {proc's parameters}
    method     : integer;           {split-stack callable?}
    nparms     : integer;        {# of target's parameters}

{declaring return parameters}

    funclen    : integer;     {length of return value     }
    funcptr    : integer;     {pointer to return value    }
    status     : xlstatus;    {how MPE XL returns warnings}
    cond_code  : ccode_type;  {how condition code returned}

{declaring local variables}

    loc_digits : shortint;
    loc_frac   : shortint;

{end Example 5-3}
```

Declaring Called External Procedures

To conclude the global declarations, you declare the external procedures called within the Switch stub procedure (see Example 5-4).

Example 5-4. DECMADD Stub, External Procedure Declarations

```
{declaring the HPSWITCHTOCM intrinsic}

PROCEDURE HPSWITCHTOCM; INTRINSIC;

{declaring the HPSETCCODE intrinsic}

PROCEDURE HPSETCCODE; INTRINSIC;
```

```
{declaring the QUIT intrinsic}

PROCEDURE QUIT; INTRINSIC;

{end Example 5-4}
```

Writing the DECMADD Stub Body

Once the declaration section is complete, it is possible to proceed to the body of the Switch stub procedure. In the body, the actual work of setting up the parameters required by the `HPSWITCHTOCM` intrinsic is done. Specifically, the body does the following:

- Initializes local variables
- Initializes the Switch intrinsic parameter variables
- Describes each target procedure parameter by doing the following:
  - Giving a pointer to the parameter's location
  - Giving the length (size) of the parameter
  - Indicating whether it is a value or a reference parameter
  - Indicating whether it is an input or an output parameter (if a reference parameter)
- Calls the `HPSWITCHTOCM` intrinsic to change modes
- Sets the condition code upon return from Switch
- Tests the MPE XL status value
- Takes action on errors

The body of the DECMADD stub procedure follows in Example 5-5. It is intended as a lab exercise to test your understanding of the operation of Switch stubs. Use the accompanying code documentation to assist you in completing the lines of code. You can check the correctness of your choices by turning to Example 5-6 immediately following this exercise where the DECMADD stub is presented in its entirety.

Example 5-5. DECMADD Stub, Stub Body

```
begin

{initializing local variables}

loc_digits :=-----------; {initialize local copy of DIGITS}
loc_frac   :=-----------; {initialize local copy of FRAC  }

{initializing SWITCH variables}

proc.p_proc_id_type   :=--------------;    {find procedure}
                                          {by name       }
proc.p_lib       := -------------; {look in PUB SL (LIB=P)}
proc.p_proc_name := '----------------';    {procedure name}
method           := -----------------;    {NOT split-stack}
                                         {callable       }
nparms           := ------;          {number of parameters}
funclen          := ------;  {length, in bytes, of function}
                            {return                      }
funcptr          := ------;     {pointer to function return}
                               {value                    }


{In the following, "describe" involves the following:}
{   1) give a pointer to the parameter's location    }
{   2) give the length (size) of the parameter       }
{   3) indicate whether value or reference parameter }
{   IF a reference parameter, THEN                   }
{   4) indicate whether input or output parameter    }


{describe OPERAND1 -- input by reference}

{determine pointer to parameter's location;           }
{addr function takes parameter as argument and returns}
{address                                              }

parms[0].pd_parmptr    := addr(-----------------);
```

```
    {determine length of parameter; sizeof function takes }
    {parameter as argument and returns number of bytes    }

    parms[0].pd_parmlen    := sizeof(--------------------);

    {reference parameter requiring byte address}

    parms[0].pd_parm_type  := -----------------------------;

    {input parameter}

    parms[0].pd_io_type    := [----------------------];
```

Example 5-5. DECMADD Stub, Stub Body, Continued

```
    {describe OPERAND2 -- input by reference}

    {determine pointer to parameter's location;           }
    {addr function takes parameter as argument and returns}
    {address                                              }

    parms[1].pd_parmptr    := addr(-----------------);

    {determine length of parameter; sizeof function takes }
    {parameter as argument and returns number of bytes    }

    parms[1].pd_parmlen    := sizeof(-------------------);

    reference parameter requiring byte address}

    parms[1].pd_parm_type  := -----------------------------;

    {input parameter}

    parms[1].pd_io_type    := [----------------------];


    {describe RESULT -- output by reference}
```

```
{determine pointer to parameter's location;           }
{addr function takes parameter as argument and returns}
{address                                              }

parms[2].pd_parmptr    := addr(-----------------);

{determine length of parameter; sizeof function takes }
{parameter as argument and returns number of bytes    }

parms[2].pd_parmlen    := sizeof(-------------------);

{reference parameter requiring byte address}

parms[2].pd_parm_type  := -----------------------------;

{output parameter}

parms[2].pd_io_type    := [----------------------];


{describe DIGITS -- input by value}

{determine pointer to parameter's location;           }
{addr function takes parameter as argument and returns}
{address                                              }

parms[3].pd_parmptr    := addr(-----------------);
```

Example 5-5. DECMADD Stub, Stub Body, Continued

```
{determine length of parameter; sizeof function takes}
{parameter as argument and returns number of bytes   }

parms[3].pd_parmlen    := sizeof(-----------------------);

{value parameter}

parms[3].pd_parm_type  := -----------------------------;
```

```
{input parameter}

parms[3].pd_io_type    := [----------------------];

{describe FRAC -- input by value}

{determine pointer to parameter's location;          }
{addr function takes parameter as argument and returns}
{address                                             }

parms[4].pd_parmptr    := addr(-----------------);

{determine length of parameter; sizeof function takes}
{parameter as argument and returns number of bytes   }

parms[4].pd_parmlen    := sizeof(-------------------);

{value parameter}

parms[4].pd_parm_type  := ----------------------------;

{input parameter}

parms[4].pd_io_type    := [----------------------];

{call the Switch intrinsic to change modes}

HPSWITCHTOCM(--------, --------, --------, --------,

           --------,  --------, --------, --------);

{set condition code upon return from Switch}

HPSETCCODE(--------);

{test MPE XL status value and set ccode if not OK}

if (status.all <> all_ok) then
```

```
      begin
         QUIT(status.info);
      end;

   end;  {stub procedure DECMADD}

   {end Example 5-5}
```

Finished DECMADD Stub

Putting all these pieces together yields the following stub procedure declaration (see Example 5-6):

Example 5-6. Finished DECMADD (NM—> CM) Stub

```
$os 'mpe/xl'$
$subprogram$
$standard_level 'ext_modcal'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$

PROGRAM Xampl56(input,output);

TYPE
   decm_number = PACKED ARRAY [1..80] of CHAR;

PROCEDURE DECMADD(var OPERAND1, OPERAND2 : decm_number;
                  var RESULT : decm_number;
                  DIGITS, FRAC : shortint);

(* SPL calling sequence:
            variable name       type          value

PROCEDURE decmadd(
            OPERAND1            BYTE ARRAY    REFERENCE
            OPERAND2            BYTE ARRAY    REFERENCE
            RESULT              BYTE ARRAY    REFERENCE
            DIGITS              INTEGER       VALUE
            FRAC)               INTEGER       VALUE    *)

const
    pidt_known  =  0;              {it's found by number}
    pidt_load   =  1;              {it's found by name  }
    pidt_plabel =  2;              {it's found by plabel}

    system_sl        =  0;
    logon_pub_sl     =  1;
    logon_group_sl   =  2;
    pub_sl           =  3;
    group_sl         =  4;

    method_normal = 0;    {not callable from split stack}
    method_split  = 1;     {callable in split-stack mode}
    method_no_copy =2;         {callable in no-copy-mode}

    parm_type_value     = 0;   {value parameter         }
    parm_type_word_ref  = 1;   {word address is required}
    parm_type_byte_ref  = 2;   {byte address is required}
```

Example 5-6. Finished DECMADD (NM—> CM) Stub, Continued

```
    ccg       = 0;           {condition code greater (>)}
    ccl       = 1;           {condition code less (<)   }
    cce       = 2;           {condition code equal (=)  }
    All_Ok    = 0;           {used in status check      }

type
    bit8               =  0..255;
    bit16              =  0..65535;
    bit8_a1            =  $ALIGNMENT 1$ bit8;
    bit16_a1           =  $ALIGNMENT 1$ bit16;
    cm_proc_name       =  packed array [1..16] of char;
    generic_buffer     =  packed array [1..65535] of char;
    scm_procedure      =  packed record
       case p_proc_id_type : bit8 of
          pidt_known : ( p_fill    : bit8_a1;
                     p_proc_id : bit16_a1 );
      pidt_load  : ( p_lib     : bit8_a1;
                    p_proc_name : cm_proc_name );
      pidt_plabel : ( p_plabel : bit16_a1);
   end;

scm_io_type = set of ( INPUT_PARM, OUTPUT_PARM );

parm_desc  =  packed record
    pd_parmptr      : globalanyptr;
    pd_parmlen      : bit16;
    pd_parm_type    : bit16;
    pd_io_type      : scm_io_type;
    end;

scm_parm_desc_array = array [0..31] of parm_desc;

ccode_type = shortint;

xlstatus =
   record
      case integer of
         0 : (all : integer);

         1 : (info : shortint;
              subsys : shortint);
      end; {record}
```

Example 5-6. Finished DECMADD (NM—> CM) Stub, Continued

```
PROCEDURE HPSWITCHTOCM; INTRINSIC;

PROCEDURE HPSETCCODE; INTRINSIC;

PROCEDURE QUIT; INTRINSIC;

{End of OUTER BLOCK GLOBAL declarations}

var
```

```
    proc          : scm_procedure;
    parms         : scm_parm_desc_array;
    method        : integer;      {split-stack callable?}
    nparms        : integer;            {# of parameters}
    funclen       : integer;
    funcptr       : integer;
    status        : xlstatus;
    cond_code     : ccode_type;

    loc_digits    : shortint;
    loc_frac      : shortint;


begin
   { Initialize local variables }
loc_digits := DIGITS;
loc_frac   := FRAC;

   { Initialize Switch variables }
proc.p_proc_id_type := pidt_load; {find procedure by name}
proc.p_lib       := pub_sl;       {look in PUB SL (LIB=P)}
proc.p_proc_name := 'DECMADD';            {procedure name}
method := method_normal;        {not split-stack callable}
nparms := 5;                        {number of parameters}
funclen := 0;       {length, in bytes, of function return}
funcptr := 0;           {pointer to function return value}

   {OPERAND1 -- input by reference}
parms[0].pd_parmptr    := addr(OPERAND1);
parms[0].pd_parmlen    := sizeof(OPERAND1);
parms[0].pd_parm_type  := parm_type_byte_ref;
parms[0].pd_io_type    := [input_parm];

   {OPERAND2 -- input by reference}
parms[1].pd_parmptr    := addr(OPERAND2);
parms[1].pd_parmlen    := sizeof(OPERAND2);
parms[1].pd_parm_type  := parm_type_byte_ref;
parms[1].pd_io_type    := [input_parm];
```

Example 5-6. Finished DECMADD (NM—> CM) Stub, Continued

```
   {RESULT -- output by reference}
parms[2].pd_parmptr    := addr(RESULT);
parms[2].pd_parmlen    := sizeof(RESULT);
parms[2].pd_parm_type  := parm_type_byte_ref;
parms[2].pd_io_type    := [output_parm];

   {DIGITS  -- input by value}
parms[3].pd_parmptr    := addr(loc_digits);
parms[3].pd_parmlen    := sizeof(DIGITS);
parms[3].pd_parm_type  := parm_type_value;
parms[3].pd_io_type    := [input_parm];

   {FRAC -- input by value}
parms[4].pd_parmptr    := addr(loc_frac);
parms[4].pd_parmlen    := sizeof(FRAC);
parms[4].pd_parm_type  := parm_type_value;
parms[4].pd_io_type  := [input_parm];

   {Execute the SWITCH}
HPSWITCHTOCM(proc, method, nparms, parms, funclen, funcptr,
           cond_code, status);

HPSETCCODE(cond_code);

if (status.all <> all_ok) then
   begin
      QUIT(status.info);
   end;

end;     {procedure DECMADD}

BEGIN {Program Outer Block Code}

END. {Program Outer Block Code}

{end Example 5-6}
```

Checklist for Writing NM—> CM Switch Stubs

The following list summarizes the steps involved in writing your own NM—> CM Switch stub procedure:

- Make the stub name identical to that of the target procedure.
- Make the names of the stub parameters identical to those of the target procedure (observing HP Pascal/XL naming conventions and restrictions).
- Make the types of the stub parameters correspond to those of the target procedure.
- Set up the parameters required by the `HPSWITCHTOCM` intrinsic.
  - Declare the necessary constants
  - Declare the necessary user-defined types
  - Declare the necessary variables
- Conclude the global declarations by declaring the procedures called within the Switch stub procedure (`HPSWITCHTOCM`, `HPSETCCODE`, and `QUIT`)
- Initialize local variables
- Initialize the Switch intrinsic parameter variables
- Describe each target procedure parameter by doing the following:
  - Give a pointer to the parameter's location
  - Give the length (size) of the parameter
  - Indicate whether it is a value or a reference parameter
  - Indicate whether it is an input or an output parameter (if a reference parameter)
- Call the `HPSWITCHTOCM` intrinsic to change modes
- Set the condition code upon return from Switch
- Test the MPE XL status value
- Take action upon errors returned

Switching to NM

To illustrate a mixed-mode Switch from Compatibility Mode (CM) to Native Mode (NM), consider the example of a Pascal/V procedure that calls an intrinsic that is available only in Native Mode. It makes this call in order to access a new NM capability.

The `HPCICOMMAND` intrinsic executes an MPE XL command programmatically. Unlike the MPE V/E-compatible `COMMAND` intrinsic, `HPCICOMMAND` allows you to execute UDC's and command files programmatically.

The syntax of the `HPCICOMMAND` intrinsic is as follows:

```
                   CA       I16    I16     I16V
         HPCICOMMAND(cmdimage,errnum,parmnum,msglevel);
```

The **cmdimage** parameter passes a required character array that represents a command string in the current language of the system. The string is limited to 280 bytes and must be terminated by a carriage return or be blank-filled up to the 280th byte.

The **errnum** parameter is a required 16-bit signed integer that is passed by reference and returns the Command Interpreter error number. If there are no errors, the value returned is 0.

The **parmnum** parameter is a required 16-bit integer that is passed by reference and indicates the nature of the error (if **errnum** <> 0). If **parmnum** is positive, this is a file system error. If **parmnum** is negative, then the absolute value of **parmnum** specifies the column number where the error occurred.

The **msglevel** parameter is a required 16-bit integer that is passed by value and indicates how error and warning messages are handled. The following list summarizes the valid values of the **msglevel** parameter and the meanings of those values:

0	All errors/warnings are printed to `$STDLIST`.
1	All Command Interpreter error/warnings are printed to `$STDLIST`.
2	No errors/warnings are printed. This is the default.

For more information on the `HPCICOMMAND` intrinsic, refer to the appropriate entry in the *MPE XL Intrinsics Reference Manual* (32650-90028).

To access the HPCICOMMAND intrinsic from CM code, you need to write a CM—> NM Switch stub and place that stub procedure in a CM Segmented Library (SL). This stub must call the appropriate Switch intrinsic (either HPSWTONMNAME or HPSWTONMPLABEL), which, in turn, invokes HPCICOMMAND. HPSWTONMNAME calls a procedure by name, while HPSWTONMPLABEL allows you to call an NM procedure by plabel. The examples in the following paragraphs illustrate a call by plabel.

Writing the CMCICommand Stub Declarations

Begin the process of writing a Switch stub with the declaration portion of the procedure.

The declaration portion includes the following:

- Stub header
- Stub parameters
- Called external procedures

Declaring the Stub Header

The following excerpt is a possible header declaration for the stub procedure to enable a call to the Native Mode HPCICOMMAND intrinsic:

```
+----------------------------------------------------------+
|                                                          |
| TYPE                                                     |
|    pac280 = PACKED ARRAY [1..280] OF CHAR;               |
|    shortint = -32768..32767;                             |
|                                                          |
| PROCEDURE CMCICommand(VAR CICmdName  : pac280;           |
|                       VAR CIErrNum   : shortint;         |
|                       VAR CIParmNum  : shortint;         |
|                           CIMsgLevel : shortint);        |
|                                                          |
+----------------------------------------------------------+
```

Declaring the Stub Parameters

The next step in writing a CM—> NM Switch stub is to set up the parameters required by the particular Switch intrinsic(s), in this instance `HPLOADNMPROC` and `HPSWTONMPLABEL`.

Consider again the Pascal/V declaration of these intrinsics:

```
FUNCTION HPSWTONMPLABEL : integer; intrinsic;

FUNCTION HPLOADNMPROC : integer; intrinsic;
```

Next come examples of Pascal/V calls to these intrinsics:

```
plabel := HPLOADNMPROC (procname, procnamelen, libname,
                        libnamelen);

returnstatus := HPSWTONMPLABEL (plabel, numparms,
                arglistarray, argdescarray, functype);
```

You call the `HPLOADNMPROC` intrinsic with four parameters. These parameters provide Switch with the following information:

- Name of the NM target procedure
- Length of the name of the NM target procedure
- Name of the library to be searched for the NM target procedure
- Length of the name of the library to be searched

The `HPSWTONMPLABEL` intrinsic is called with five parameters, providing Switch with this information:

- Plabel of the NM routine (as returned by the `HPLOADNMPROC` intrinsic)
- Number of parameters in the NM routine
- Parameter list (value parameters represented by their values; reference parameters, by their DB-relative addresses)
- Parameter description list (specifying the type of the parameters)
- Type of the functional return value (if any)

Setting up these parameters involves two stages:

1. Declaring the necessary user-defined types
2. Declaring the necessary variables

Declaring User-Defined Types. Several user-declared types are necessary to set up the stub procedure variables. These include the following:

- String type to represent procedure and library names
- String type to represent MPE XL command or UDC names
- Status Type
- Integer subrange type to represent error numbers, the number of parameters, message levels, the length of character strings, and function types
- Array of integer subrange type values to represent and describe the parameters of the target procedure

The type declaration section for the CMCICommand stub procedure follows in Example 5-7:

Example 5-7. CMCICommand Stub, Type Declarations

```
{Type Declarations}

TYPE
   shortint = -32768..32767;
   shr_ary32 = packed array [1..32] of shortint;
   pac16 = packed array [1..16] of char;
   pac280 = packed array [1..280] of char;

{end Example 5-7}
```

Declaring Variables. You declare two groups of variables in the variable declaration section of the stub:

- Nine variables required by the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics
- Any local variables

The variable declaration section for the CMCICommand stub procedure follows in Example 5-8:

Example 5-8. CMCICommand Stub, Variable Declarations

```
{Global variable declarations}

VAR

{Parameters passed to HPCICOMMAND via Switch}

   CmdName  : pac280;
   ErrNum   : shortint;
   ParmNum  : shortint;
   MsgLevel : shortint;

{Stub procedure variable declarations}

VAR

{HPLOADNMPROC intrinsic parameters}

   proc_name : pac16;    {name of NM routine}
   proc_len : shortint;  {length of target's name}
   lib_nam : pac16;      {NM library to search for target}
   lib_len : shortint;   {length of NM library name}

{HPSWTONMPLABEL intrinsic parameters}

   proc   : integer;     {plabel of NM routine}
   nparms : shortint;    {number of target parameters}
   arglist : shr_ary32;  {parameter list}
   argdesc : shr_ary32;  {parameter description list}
   fct_typ : shortint;   {functional return type, if any}

{Assigned functional return}

   rtn_st : integer;
```

```
{end Example 5-8}
```

Declaring Called External Procedures

You must also declare the procedures called within the Switch stub procedure (see Example 5-9).

Example 5-9. CMCICommand Stub, External Procedure Declarations

```
{Intrinsic procedure declarations}

FUNCTION HPSWTONMNAME : integer; intrinsic;

FUNCTION HPSWTONMPLABEL : integer; intrinsic;

FUNCTION HPLOADNMPROC : integer; intrinsic;

{end Example 5-9}
```

Writing the CMCICommand Stub Body

Once the declaration section is complete, proceed to the body of the Switch stub procedure. In the body, the actual work of setting up the parameters required by the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics is done. Specifically, the body does the following:

- Initializes local variables
- Initializes the Switch intrinsic parameter variables as follows:
  - Designates names as string values
  - Designates the lengths of the strings
  - Designates the number of target procedures
  - Designates the type of the target's functional return
  - Initializes arrays to represent and describe target procedure parameters
- Calls the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics
- Copies local variables back to formal parameters
- Tests the status value
- Takes action on errors

The body of the CMCICommand stub procedure follows in Example 5-10. It is intended as a lab exercise to test your understanding of the operation of Switch stubs. Use the accompanying code documentation to assist you in completing the lines of code. You can check the correctness of your choices by turning to Example 5-11 immediately following this exercise where the CMCICommand stub is presented in its entirety.

Example 5-10. CMCICommand Stub, Stub Body

```
BEGIN {Stub procedure CMCICommand}

{Initializations}

   proc_name := '----------------';  {name of target}
   proc_len  := ----;         {length of target name}
   lib_name  := '----------------';  {name of library}
   lib_len   := ----;        {length of library name}

   nparms    := ---;  {nparms governs how many types are}
                      {searched for in argdesc;         }
                      {4 parms plus extensible_gateway  }
                      {HPCICOMMAND is declared          }
                      {$OPTION 'EXTENSIBLE_GATEWAY$     }

   arglist[1]  := --; {extensible gateway mask is 32 bits}
   arglist[2]  := --; {of anything, all 0 bits works ok! }

   arglist[3]  := baddress(------------);
               {reference parameter passed by address;   }
               {take byte address of variable name buffer}

   arglist[4]  := waddress(-----------);
                    {reference parameter passed by address;}
                    {take word address of error number parm}

   arglist[5]   := waddress(------------);
                    {reference parameter passed by address;}
                    {take word address of parameter number }
                    {parm                                  }

   arglist[6]  := -------------;
                  {value parameter represented by its value}

   argdesc[1]  := ---; {32-bit word value for gateway mask}
   argdesc[2]  := ---; {byte pointer for arglist[3]}
   argdesc[3]  := ---; {word pointer for arglist[4]}
```

```
argdesc[4] := ---; {word pointer for arglist[5]}
argdesc[5] := ---; {shortint value parameter}

fct_typ := ---; {This is an NM procedure, not a      }
           {function.  If it was an NM function, the }
           {return value would come back in          }
           {arglist[1..n] where n is the length of   }
           {the value in 16-bit words.               }
```

Example 5-10. CMCICommand Stub, Stub Body, Continued

```
{HPLOADNMPROC intrinsic call -- 4 parameters}

   proc := HPLOADNMPROC
                         (------------,  {target name}
                          ------------,  {length of target }
                                         {name             }
                          ------------,  {library name}
                          ------------); {length of library}
                                         {name             }

{HPSWTONMPLABEL intrinsic call -- 5 parameters}

   rtn_st := HPSWTONMPLABEL
                           (------------, {returned plabel}
                            ------------, {number of target}
                                          {parameters      }
                            ------------, {target argument }
                                          {list array      }
                            ------------, {target argument }
                                         {description array}
                            ------------);  {type of target}
                                         {functional return}

END;  {Stub procedure CMCICommand}

{end Example 5-10}
```

Finished CMCICommand Stub

Putting all these pieces together yields the following stub procedure declaration (see Example 5-11):

Example 5-11. Finished CMCICommand (CM—> NM) Stub

```
{ XAMPL511 -- example Switch to NM by plabel }
$standard_level "HP3000"$
{$subprogram$} {uncomment this to make an RBM for your SL}
$uslinit$

PROGRAM XAMPL511(input, output);

{Type Declarations}

TYPE
   shortint = -32768..32767;
   shr_ary32 = packed array [1..32] of shortint;
   pac16 = packed array [1..16] of char;
   pac280 = packed array [1..280] of char;

{Global variable declarations}

VAR

{Parameters passed to HPCICOMMAND via Switch}

   CmdName  : pac280;
   ErrNum   : shortint;
   ParmNum  : shortint;
   MsgLevel : shortint;

{Intrinsic procedure declarations}

FUNCTION HPSWTONMNAME : integer; intrinsic;

FUNCTION HPSWTONMPLABEL : integer; intrinsic;
FUNCTION HPLOADNMPROC : integer; intrinsic;
```

Example 5-11. Finished CMCICommand (CM—> NM) Stub, Continued

```
{Stub procedure declaration}

PROCEDURE CMCICommand(VAR CICmdName  : pac280;
                      VAR CIErrNum   : shortint;
                      VAR CIParmNum  : shortint;
                          CIMsgLevel : shortint);

VAR

{HPLOADNMPROC intrinsic parameters}

   proc_name : pac16;    {name of NM routine}
   proc_len : shortint;  {length of target's name}
   lib_name : pac16;     {NM library to search for target}
   lib_len : shortint;   {length of NM library name}

{HPSWTONMPLABEL intrinsic parameters}

   proc   : integer;     {plabel of NM routine}
   nparms : shortint;    {number of target parameters}
   arglist : shr_ary32;  {parameter list}
   argdesc : shr_ary32;  {parameter description list}
   fct_typ : shortint;   {functional return type, if any}

{Assigned functional return}

   rtn_st : integer;

BEGIN {Stub procedure CMCICommand}

{Initializations}

   proc_name := 'HPCICOMMAND     ';
   proc_len  := 11;
   lib_name  := 'NL.PUB.SYS      ';
   lib_len   := 10;
```

```
nparms    := 5;    {nparms governs how many types are}
                   {searched for in argdesc;         }
                   {4 parms plus extensible_gateway  }
                   {HPCICOMMAND is declared          }
                   {$OPTION 'EXTENSIBLE_GATEWAY$     }
```

Example 5-11. Finished CMCICommand (CM—> NM) Stub, Continued

```
    arglist[1] := 0;  {extensible gateway mask is 32 bits}
    arglist[2] := 0;  {of anything, all 0 bits works ok! }

    arglist[3] := baddress(CICmdName);
               {reference parameter passed by address;   }
               {take byte address of variable name buffer}

    arglist[4] := waddress(CIErrNum);
                    {reference parameter passed by address;}
                    {take word address of error number parm}

    arglist[5]  := waddress(CIParmNum);
                    {reference parameter passed by address;}
                    {take word address of parameter number }
                    {parm                                  }

    arglist[6] := CIMsgLevel;
                  {value parameter represented by its value}

    argdesc[1] := 03; {32-bit word value for gateway mask}
    argdesc[2] := 05; {byte pointer for arglist[3]}
    argdesc[3] := 06; {word pointer for arglist[4]}
    argdesc[4] := 06; {word pointer for arglist[5]}
    argdesc[5] := 02; {shortint value parameter}

    fct_typ := 00; {This is an NM procedure, not a       }
               {function.  If it was an NM function, the }
               {return value would come back in          }
               {arglist[1..n] where n is the length of   }
               {the value in 16-bit words.               }

 {HPLOADNMPROC intrinsic call}

    proc := HPLOADNMPROC
                      (proc_name,
                       proc_len,
                       lib_name,
                        lib_len);

{HPSWTONMPLABEL intrinsic call}

   rtn_st := HPSWTONMPLABEL
                         (proc,
                          nparms,
                          arglist,
                          argdesc,
                          fct_typ);

END;  {Stub procedure CMCICommand}
```

Example 5-11. Finished CMCICommand (CM—> NM) Stub, Continued

```
BEGIN  {outer block}

         .
         .
         .
   CmdName := '                ';
   ErrNum := 0;
   ParmNum := 0;
   MsgLevel := 0;

   CMCICommand(CmdName, ErrNum, ParmNum, MsgLevel);
         .
         .
         .

END.  {outer block}

{end Example 5-11}
```

Checklist for Writing CM—> NM Switch Stubs

The following list summarizes the steps involved in writing your own CM—> NM Switch stub procedure:

- Set up the parameters required by the `HPSWTONMNAME` or `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics
  - Declare the necessary constants
  - Declare the necessary user-defined types
  - Declare the necessary variables
- Conclude the global declarations by declaring the procedures called within the Switch stub procedure
- Initialize local variables
- Initialize the Switch intrinsic parameter variables
- Call the `HPSWTONMNAME` intrinsic or the `HPLOADNMPROC` and `HPSWTONMPLABEL` intrinsics to change modes
- Test the status value
- Take action upon errors returned

A
SWAT Warning Messages

The following are the warning messages that the Switch Assist Tool issues.

0001	The file specified already exists.
0002	The file specified already exists and has a lockword.
0003	Invalid character found.
0004	Unknown problem while checking filename.
0005	The first character must be ALPHABETIC.
0006	The function key pressed is not allowed here.
0007	The filename for the generated source is required.
0008	The name of the target procedure is required.
0009	Duplicate parameter names are not allowed.
0010	Only one library location may be specified.
0011	One library location must be specified.
0012	Not used.
0013	Not used.
0014	Only one functional return option may be selected.
0015	One functional return option must be selected.
0016	Only one privilege level option may be selected.
0017	One privilege level must be selected.
0018	Only one condition code return option may be selected.
0019	One condition code return option must be selected.
0020	Parameter name field cannot be left blank here.
0021	Only one addressing method may be specified.
0022	One addressing method must be selected.
0023	Only one input/output direction may be selected.
0024	One input/output direction must be selected.
0025	Parameters passed by value must be input only.
0026	Only one data type may be selected.
0027	One data type must be selected.
0028	Arrays cannot be passed by VALUE.
0029	Only one array specification may be selected.
0030	One array specification must be selected.
0031	Length of array cannot be specified by both a constant and a parameter.

0032	Length of array must be specified.
0033	Input must be a number in the range of 1 to 65535.
0034	Only one array length usage may be selected.
0035	One array length usage must be selected.
0036	The indirect length variable must not be an array.
0037	The indirect length variable is not defined.
0038	The MAIN screen has not been entered successfully.
0039	The PROCINFO screen has not been entered successfully.
0040	Variable has undefined indirect length parameter.
0041	Variable's indirect length parameter is output only.
0042	Indirect parameter can't be logical with negative=bytes rule.
0043	Indirect parameter must be integer, logical, or double data type.
0044	The following parameters need more attention.
0045	Press F2 when ready to begin generating code.
0046	The ENTER key is not allowed here.
0047	You must attend to above items before code generation can begin.
0048	Length of byte array must be in the range 1 to 65535 elements.
0049	Length of integer array must be in the range 1 to 32767 elements.
0050	Length of logical array must be in the range 1 to 32767 elements.
0051	Length of double array must be in the range 1 to 16383 elements.
0052	Length of real array must be in the range 1 to 16383 elements.
0053	Length of long array must be in the range 1 to 8191 elements.

B

SWAT Fatal Error Messages

The following are the fatal error messages that can appear on an abort screen if the Switch Assist Tool encounters an irrecoverable error.

0001	Fatal error returned from the VGETFIELD intrinsic.
0002	Fatal error returned from the VSETERROR intrinsic.
0003	Fatal error returned from the VPUTFIELD intrinsic.
0004	Fatal error returned from the VPUTWINDOW intrinsic.
0005	Fatal error returned from the VGETBUFFER intrinsic.
0006	Fatal error returned from the VPUTBUFFER intrinsic.
0007	Fatal error returned from the VSHOWFORM intrinsic.
0008	Fatal error returned from the VREADFIELDS intrinsic.
0009	Fatal error returned from the VGETNEXTFORM intrinsic.
0010	Fatal error returned from the VINITFORM intrinsic.
0011	Fatal error returned from the VSETKEYLABEL intrinsic.
0012	Fatal error returned from the VFIELDEDITS intrinsic.
0013	Fatal error returned from the VERRMSG intrinsic.
0014	Internal inconsistencies found when attempting to provide help.
0015	Unknown target language found when transferring to second screen.
0016	Unknown target language found when transferring to third screen.
0017	The next screen found is not a valid screen number.
0018	CCL was returned when attempting to open the new source file.
0019	Not used.
0020	Impossible ccode returned from FOPEN, not CCE or CCL?
0021	Impossible ccode returned from FCLOSE, not CCE or CCL?
0022	CCL returned when re-fopening the new source file.
0023	Impossible ccode returned when re-fopening, not CCE or CCL?
0024	An irrecoverable error occurred when opening new source file.
0025	Passed parameter is neither by VALUE, or by REFERENCE?
0026	The data type of the parameter is not one of the possible types.
0027	*** UNUSED ***
0028	The FUNCTIONAL return type is not a valid type.
0029	This program may not be run in a BATCH job. It uses VPLUS/3000.
0030	A fatal error has occurred while accessing the message catalog.
0031	Found bad data type when building local copies of VALUE parms.
0032	Found bad FUNCTIONAL return type when creating local variable.
0033	Found bad data type when calculating element size in bytes.
0034	The direction does not specify NM -> CM, or CM -> NM.
0035	Fatal error when opening the msg catalog file SWATCAT.PUB.SYS.
0036	Fatal error while opening SWATCAT.PUB.SYS.
0037	Fatal error returned by VOPENTERM intrinsic.
0038	Fatal error returned by VOPENFORMF intrinsic.

C

SWAT Progress Messages

The following are the progress messages that appear on the PROGRESS screen while the Switch Assist Tool is generating Switch stub source code.

0001	Adjusting variable names to PASCAL standards.
0002	Generating outer block global declarations.
0003	Generating procedure declaration for the STUB procedure.
0004	Generating parameter declarations for the STUB procedure.
0005	Closing out the STUB procedure declarations.
0006	Setting up local constant values within the STUB procedure.
0007	Generating the STUB procedure "BEGIN" statement, with comments.
0008	Generating declarations for a local copy of any VALUE parameters.
0009	Generating local variable for a local copy of functional return.
0010	Generating local variables for use by the STUB procedure.
0011	Building target procedure specifications for the SWITCH call.
0012	Generating misc. initialization code.
0013	Generating code to handle a functional return value.
0014	Generating code to do initializations.
0015	Generating code to move value parms to STUB local copies.
0016	Building the parameter descriptor array for the SWITCH call.
0017	Generating the actual call to the SWITCH procedure.
0018	Generating code to pass the functional return back to caller.
0019	Generating code to handle the condition code return value.
0020	Generating the main body of the STUB procedure.
0021	Generating the STUB procedure's bottom outer block.

D

TDP Use File for Concatenating Switch Stubs

```
:comment |-------------------------------------------------|
:comment |                                                 |
:comment | This file is meant to be used as a 'USE' file   |
:comment | with TDP/3000 to concatenate multiple STUB      |
:comment | procedure source files into a single file.  To |
:comment | use this utility, perform the following steps: |
:comment |                                                 |
:comment | :RUN TDP.PUB.SYS                                |
:comment | /USE CATTDP                                     |
:comment |  --- Then simply answer the prompts as they -- |
:comment |  --- are presented.                         -- |
:comment | /EXIT                                           |
:comment |                                                 |
:comment | * NOTE *                                        |
:comment | These comment lines DO NOT HAVE TO BE REMOVED  |
:comment | for this TDP 'USE' file to function correctly. |
:comment |                                                 |
:comment |-------------------------------------------------|
set shorterror
set quiet
q " ";q " ";q " ";q " ";q " "
q " Switch Assist Tool (SWAT) STUB concatenation utility "
q " ";q " "
q "In the following dialog, enter the information"
q "requested when prompted."
q " ";q " "
q "Enter first STUB FILE name below.  This file may be a"
q "file as generated by SWAT, or a concatenated STUB file"
```

```
q "created during a previous execution of this utility."
q " "
zp="First STUB FILE : "
q " "
q "Bringing in file (z)"
q " "
text zq " "
verify total
q " "
q "Extracting top outer block, and first STUB"
```

```
q " "
holdq first/last-3
@L1 q " ";q " "
q "Enter ANOTHER STUB FILE name below.  This file may be a"
q "file as generated by SWAT, or a concatenated STUB file"
q "created during a previous execution of this utility."
q " "
zp="Enter ANOTHER STUB file name : "
q " "
q "Bringing in file (z)"
q " "
t zq " "
verify total
q " "
q "Skipping over TOP OUTER BLOCK"
q " "
findq "$PAGE$",NOTEXT
q "Extracting current STUB procedure"
holdq */last-3,APPEND
q " "
@IF "Concatenating another STUB? (Y/N) : " THEN GO TO L1
q " ";q " "
q "Appending bottom outer block code"
holdq last-2/last,APPEND
q " "
q "Rebuilding work file"
dq all
addq 1,HOLDQ
q " ";q " "
zp="Enter NEW file name to contain concatenated STUBS : "
k z,unn
q " ";q " ";q " "
q "STUB concatenation complete"
q " "
set display
set longerror
```

E

EDIT Use File for Concatenating Switch Stubs

```
:comment  - - - - - - - - - - - - - - - - - -  - - - - - - -
:comment |                                                  |
:comment | This file is meant to be used as a 'USE' file   |
:comment | with EDITOR/3000 to concatenate multiple STUB   |
:comment | procedure source files into a single file.  To |
:comment | use this utility, perform the following steps: |
:comment |                                                  |
:comment | :EDITOR                                          |
:comment | /USE CATEDIT                                     |
:comment |  --- Then simply answer the prompts as they -- |
:comment |  --- are presented.                         -- |
:comment | /EXIT                                            |
:comment |                                                  |
:comment | * NOTE *                                         |
:comment | These comment lines DO NOT HAVE TO BE REMOVED   |
:comment | for this EDITOR 'USE' file to function          |
:comment | correctly.                                       |
:comment |                                                  |
set short - - - - - - - - - - - - - - - - - - - - - - - - -
set quiet
q " ";q " ";q " ";q " ";q " "
q " Switch Assist Tool (SWAT) STUB concatenation utility "
q " ";q " "
q "In the following dialog, you must enter the requested"
q "information following the EDITOR 'ENTER Z=' prompt. "
q " ";q " "
q "Enter first STUB FILE name below.  This file may be a"
q "file as generated by SWAT, or a concatenated STUB file"
```

```
q "created during a previous execution of this utility."
q " "
z=
q " "
q "Bringing in file (z)"
q " "
text z
```

```
q " "
verify total
q " "
q "Extracting top outer block, and first STUB"
q " "
holdq first/last-3
set batch
z=
//
set poll
WHILE FLAG
  BEGINQ
  q " ";q " "
  q "Enter ANOTHER STUB FILE name below.  This file may be"
  q "a file as generated by SWAT, or a concatenated STUB"
  q "file created during a previous execution of this"
  q "utility."
  q " "
  q "** If there are NO MORE STUB FILES to concatenate **"
  q "** press CTRL-Y to complete the concatenation and **"
  q "** generate a new file.                           **"
  q " "
  z=
  q " "
  q "Bringing in file (z)"
  q " "
  t z
  q " "
  verify total
  q " "
  q "Skipping over TOP OUTER BLOCK"
  q " "
  findq "$PAGE$"
  q "Extracting current STUB procedure"
  holdq */last-3,APPEND
  q " "
  END
YES
q "Appending bottom outer block code"
holdq last-2/last,APPEND
q " "
```

```
q "Rebuilding work file"
dq all
add,holdq,now
q "Enter NEW file name to contain the concatenated STUB"
q "procedures."
q " "
z=
k z,unn
q " ";q " ";q " "
q "STUB concatenation complete"
q " "
set long
set display
```