900 Series HP 3000 Computer Systems

MPE/iX Architected Interface Facility: Procedure Exits Reference Manual

HEWLETT PACKARD

HP Part No. 36429-90001
Printed in U.S.A. 1994

Fourth Edition
E0494

Architected Interfaces Notice

Copyright © 1994 by Hewlett-Packard Company

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Restricted Rights Legend

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	December 1990	A.01.01
Second Edition	June 1992	B.40.00
Third Edition	October 1992	C.45.00
Fourth Edition	March 1994	C.50.00

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s not based on PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

MPE/iX Architected Interface Facility: Procedure Exits Reference Manual (36429-90001) is written for the experienced programmer.

This manual is organized into the following chapters and appendixes:

Chapter 1 **Introduction** contains an introductory overview of architected interfaces in general and procedure exits architected interfaces (AIFs) in particular, as well as installation procedures.

Chapter 2 **Overview of Procedure Exits** defines concepts and terminology and presents a comprehensive description of how procedure exit AIFs work.

Chapter 3 **Procedure Exits Architected Interfaces** describes the various types of architected interfaces available.

Chapter 4 **Architected Interface Programming Considerations** lists data type naming conventions and the Architected Interface Facility error management strategy.

Chapter 5 **Architected Interface Descriptions** provides detailed descriptions and syntax of the procedure exits architected interfaces.

Chapter 6 **Architected Interface Examples** provides programming examples in the Pascal and C languages. These examples show how to use the procedure exits architected interfaces.

Appendix A **AIF Status Messages** contains a list of all the error messages returned by architected interfaces.

Appendix B **AIF Data Structures** contains a list of all the data structures used in the architected interfaces.

Appendix C **Handler Coding Requirements** provides information on how you must code procedure exits handlers.

Appendix D **Glossary** provides definitions of terms used in this manual.

Conventions

UPPERCASE

In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command     Command     COMMAND
```

It cannot, however, be entered as:

```
comm        com_mand    comamnd
```

italics

In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

```
COMMAND filename
```

bold italics

In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

```
COMMAND(filename)
```

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

```
(filename):(filename)
```

underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

```
Do you want to continue? >> yes
```

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either ON or OFF:

```
COMMAND { ON  }
        { OFF }
```

[ ]

In a syntax statement, brackets enclose optional elements. In the following example, OPTION can be omitted:

```
COMMAND filename [OPTION]
```

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select OPTION or *parameter* or neither. The elements cannot be repeated.

```
COMMAND filename [ OPTION    ]
                 [ parameter ]
```

Conventions (continued)

`[ ... ]`

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

`[,`*parameter*`][...]`

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

`[`*parameter*`][,...]`

`| ... |`

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select `A`, `AB`, `BA`, or `B`. The elements cannot be repeated.

$$\left\{ \begin{matrix} \texttt{A} \\ \texttt{B} \end{matrix} \right\} \texttt{| ... |}$$

...

In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

Δ

In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

`(`*parameter*`)Δ(`*parameter*`)`

The symbol indicates a key on the keyboard. For example, RETURN represents the carriage return key or Shift represents the shift key.

CTRL *character*

CTRL *character* indicates a control character. For example, CTRL Y means that you press the control key and the Y key simultaneously.

Contents

Tables

1

Introduction

The MPE/iX Architected Interface Facility provides reliable, high-performance development tools for 900 Series HP 3000 system management suppliers. The MPE/iX Architected Interface Facility provides specialized procedures, architected interfaces (AIFs), for use by software suppliers and internal and external solutions creators. AIFs provide easy and high-performance access, manipulation, or interception of Hewlett-Packard proprietary operating system and subsystem processes.

AIFs are a software layer between non-operating system software and internals, providing controlled access to MPE/iX internal functionality and data structures. AIFs, executing at user privileged mode (PM), provide a window into MPE/iX internal operations.

AIFs do not supply a direct image of MPE/iX internals, but rather abstract the operating system structures. For example, a management utility needs to know everything about a specific session but does not need to know the format of the internal structure's contents. This abstraction gives AIF users independence from MPE/iX details and most implementation changes.

Note AIFs will change to reflect changes to MPE/iX internals.

The only programming languages supported by AIFs are C and Pascal.

AIFs are available only for use with the MPE/iX operating system.

Intended Use for Architected Interfaces

Hewlett-Packard provides two layers of programmatic access to the MPE/iX operating system, allowing software suppliers to select the layer that best meets their needs:

- AIFs provide high-performance access.
- System intrinsics provide totally secure access.

In the past, although information has been available through intrinsics, software suppliers have used privileged mode to meet performance needs, risking data integrity and system reliability problems possible with the use of privileged mode.

This concern for performance is addressed in the AIF design, which increases performance while minimizing error checking. AIFs are faster and more functional than intrinsics, while providing a higher degree of data integrity and system reliability than privileged mode access.

Architected Interface Facility products provide supported AIFs without the commitment to backward compatibility as with system intrinsics. For example, many MPE/iX intrinsics were included just to ensure backward compatibility with MPE V. New MPE/iX intrinsics provide the same backward compatibility over the life of MPE/iX. On the other hand, AIFs may change over the life of MPE/iX to reflect changes to system internals. AIF design will be consistent over time, but data returned or the functions provided will change as underlying operating system data structures and functionality change.

Caution

Any use of privileged mode should be carefully considered because the normal checks and limitations that apply to standard users are bypassed in privileged mode. A privileged mode program can destroy file integrity and the MPE/iX operating system software. 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/iX operating system.

Who Uses Architected Interfaces?

The primary audience of the Architected Interface Facility is third-party developers outside of Hewlett-Packard. The secondary audience is Hewlett-Packard internal operating system, subsystem, and application developers.

MPE/iX Architected Interface Facility products are available for purchase by any third party developer. Although other audiences can benefit from using AIFs, they should have a sufficient level of technical sophistication.

Architected Interface Facility: Procedure Exits Features

The Architected Interface Facility: Procedure Exits product enables you to replace or augment system functionality on MPE/iX. Software solutions may be accomplished through run time interception of MPE/iX procedures residing in `NL.PUB.SYS` or other system libraries. It does this by letting you specify certain procedures to be executed in place of, or in addition to, existing procedures within system or user code/applications in both compatibility mode (CM) and native mode (NM). This specification can apply to anywhere from a single process to all processes on the system.

Note

An overview of Architected Interface Facility: Procedure Exits concepts and terminology is presented in the following chapter, and may be useful in better understanding the product descriptions that follow.

Intercepting Procedures

The Architected Interface Facility: Procedure Exits product allows the user to specify a user procedure, or handler routine, to be executed either prior to, or upon completion of, execution of a target procedure. Both the handler and the target procedures may reside in any library, but may not reside in the same library.

For example, any calls to the `FOPEN` intrinsic in `NL.PUB.SYS` could be intercepted with a user procedure `FOO` in a user library, `FOOXL`. If `FOO` is defined by the user as an invocation handler, then it will be executed before the `FOPEN` intrinsic is executed.

Stubbing Procedures

This feature allows the user to skip execution of the target procedure, and only execute the handler procedures specified. This effectively replaces the target procedure with one or more handler procedures. For example, a security product may wish to perform its own security checks upon logon and not perform the MPE security check. It could simply intercept the logon routine that does the MPE security check, stub it out, and execute its own checks instead.

Intercepting Internal and External Procedure Calls

Procedure calls can be classified into two categories. These are inter-SOM calls (external) and intra-SOM calls (internal). Intercepting all calls to a procedure implies that both internal and external calls are intercepted. The Architected Interface Facility: Procedure Exits product will ensure that both internal and external calls are intercepted for a fixed set of procedures in the NL, XL, and the SL (in CM only). This set of procedures may change from release to release, and users should check the README.PE.SYS file for details. For all other procedures, only external calls will be intercepted. Thus, if users wish to intercept all calls to a user procedure, then that procedure should only be called externally. This can be ensured by placing the procedure in a module separate from any of its callers or by dynamically calling the procedure, for example.

Caution

Hewlett-Packard support for internal procedures, both instrumented and non-instrumented, extends only to the Architected Interface Facility: Procedure Exits software. Hewlett-Packard does not guarantee nor provide support for the internal procedures. An internal procedure may function differently from release to release and may even disappear over time. To ensure that your application software is not adversely affected by such changes to internal procedures, it is recommended that you thoroughly test your application with each release of the MPE/iX operating system. If you require access to a pre-release version of the operating system for this testing, please contact your Hewlett-Packard support representative.

Intercepting CM Procedures

The Architected Interface Facility: Procedure Exits product allows users to intercept Compatibility Mode procedure calls as well. Users must **instrument** the procedures by creating a Native Mode dummy procedure, or stub, for the Compatibility Mode procedure. The Compatibility Mode procedure must then be modified to first call `PEINST` (located in `SL.PUB.SYS`) that will set up the environment for interception and call the appropriate handlers.

The Architected Interface Facility: Procedure Exits product requires that all handler procedures be implemented in Native Mode. Thus, in order for a user to access the parameters of a Compatibility Mode target procedure from any handler, the user must be able to convert Compatibility Mode addresses to the corresponding Native Mode virtual addresses. The procedure exits AIF `AIFCONVADDR` gives users this capability. (`AIFCONVADDR` is also available through the Architected Interface Facility: Operating System product.)

Intercepting Bindings

The Architected Interface Facility: Procedure Exits product allows users to define a handler/target binding for any existing process on the system.

Inheriting and Bequeathing Bindings

You may specify that bindings for a process are to be bequeathed (inherited) to any of its child process(es), existing or future. Thus, a binding can be in effect for an entire process tree. You can define a binding that will be in effect for every process (existing or future) on the system by calling `PEARM` with PIN=1 and BEQUEATHE=TRUE.

Multiple Target Handlers

The Architected Interface Facility: Procedure Exits product allows the coexistence of multiple handler procedures for the same target procedure. This means that different vendors with different products will be able to intercept the same target routine with their own unique handler procedures.

Prioritizing Handlers

A priority scheme executes the invocation handlers for the target in descending priority order, and executes the termination handlers in ascending priority order. The Architected Interface Facility: Procedure Exits product allows users to specify a priority for their handlers, with options for maximum and minimum priorities available. If not explicitly specified, then the next lowest priority is assigned.

Access to Parameters

Users of the Architected Interface Facility: Procedure Exits product have access to all parameters that are passed into the target procedure being intercepted, and all parameters returned by the target procedure, including functional result. Thus, a termination handler, for example, could examine a status parameter returned by the target procedure, and take appropriate action depending on whether a good or bad status is returned.

Dynamic Binding and Unbinding of Handlers

The Architected Interface Facility: Procedure Exits product binds and unbinds handler routines to target routines dynamically, without the need for rebooting or relinking the system. Binding/unbinding affects all processes currently running, as well as those subsequently created.

Locking and Unlocking

The Architected Interface Facility: Procedure Exits product used to provide a way to obtain exclusive access to Procedure Exits data and functions. However, due to locking strategy issues, this functionality has been removed in MPE/iX release 5.0.

Note

The Intrinsics `PELOCK` and `PEUNLOCK` have been left for compatibility reasons, but do NOT provide any functionality.

Product Components

The Architected Interface Facility: Procedure Exits product includes the following components:

`PEINTR.PE.SYS`	An intrinsic file containing the calling sequences for procedure exits AIFs.
`PEXL.PUB.SYS`	An executable library (XL) file containing the executable code for the procedure exits arming AIFs `PEARM` and `PEDISARM`.
`PEUTIL.PUB.SYS`	An interactive utility (located in `PUB.SYS` on all 900 Series HP 3000 computer systems MPE/iX release 3.0 and later) that provides system managers at customer sites the capability of allowing or disallowing the execution of procedure exits AIFs on a particular system.
`README.PE.SYS`	A "Read Me" file explaining the Architected Interface Facility: Procedure Exits product and an alphabetical list of instrumented procedures and their descriptions.

Installing Procedure Exits Architected Interfaces

The Architected Interface Facility: Procedure Exits product includes the following files:

- `PEINTR.PE.SYS`
- `PEXL.PUB.SYS`

PEINTR

`PEINTR` is a binary file that contains the intrinsic definitions of all procedure exits AIFs. Use `PEINTR` in your program to declare procedure exits AIFs.

Following is an example of Pascal code that enables the HP Pascal/iX compiler to locate the procedure exits AIF intrinsic file PEINTR:

```
PROGRAM foo;
PROCEDURE intrinsic_foo; INTRINSIC; { Compiler looks for the procedure }
                                    { intrinsic_foo in the intrinsic   }
                                    { file  SYSINTR.PUB.SYS by default.}

$SYSINTR 'PEINTR.PE.SYS'$          { Switches the intrinsic file      }
PROCEDURE aif_foo; INTRINSIC;       {Compiler looks in PEINTR.PE.SYS  }
begin
end.
```

Following is an example of C code that enables the HP C/iX compiler to locate the procedure exits AIF intrinsic file PEINTR:

```
#pragma intrinsic_file "peintr.pe.sys"
```

PEXL PEXL.PUB.SYS is an executable library (XL) file containing the executable code for the Architected Interface Facility: Procedure Exits product. It is present on all 900 Series HP 3000 computer systems MPE/iX release 3.0 and later. In order to use procedure exits AIF, you must link the PEXL library file with your program file using options available either at link time or at run time.

Note PEXL.PUB.SYS must be the last library specified in the executable library list.

Linking PEXL at Link Time

You can link PEXL to your program file at link time using the XL= option of the LINK command. For example:

```
:LINKEDIT
linkedit>LINK FROM=YMYPROG; TO=MYPROG;CAP=PM;XL='XL1, ... XLN,PEXL.PUB.SYS'
linkedit>EXIT
:
```

Linking PEXL at Run Time

You can link PEXL to your program file at run time, using the XL= option of the RUN command. For example:

```
:RUN MYPROG;XL='XL1, ... XLN,PEXL.PUB.SYS'
```

How to Ship Products That Use Procedure Exits Architected Interface

Before you ship your product to customer sites, you should take into account the following considerations:

- Installing your Architected Interface Facility: Procedure Exits user ID onto the customer system.
- Providing documentation on the PEUTIL utility to allow system managers to enable and disable execution of procedure exits on their system.

Installing Your User ID

In order to ship code using procedure exits AIF to customer sites, `AIFGLOBINSTALL` must be executed on all systems containing code that calls procedure exits AIF (for example, your application). It should be executed once per installation. However, it can be executed each time your application is run without side effects. Your application must execute `AIFGLOBINSTALL` prior to calling any other procedure exits AIF intrinsic.

`AIFGLOBINSTALL` installs on the target system the unique user ID that is assigned to you by Hewlett-Packard at the time of purchase of the Architected Interface Facility: Procedure Exits product.

Note

It is strongly recommended that only the user ID provided by Hewlett-Packard be installed.

`AIFGLOBINSTALL` will fail if not enough disk space is located on LDEV 1. If this occurs, you must create additional free space on LDEV 1 before attempting to re-execute code that contains the call to `AIFGLOBINSTALL`.

Providing PEUTIL Instructions

If you want to provide your customer with the ability to enable or disable procedure exits on their system, you should provide instructions on how your customers can use PEUTIL to enable and disable execution of procedure exits on their system. Chapter 3 includes instructions for using PEUTIL.

2

Overview of Procedure Exits

This chapter provides an overview of the Architected Interface Facility: Procedure Exits product. It first lays a groundwork of understanding by explaining the concepts behind Architected Interface Facility: Procedure Exits, and then follows up with a detailed description of how the product actually works.

Overview

The Architected Interface Facility: Procedure Exits product allows users to replace, or enhance, procedures within the operating system or user applications with their own procedures. Through a programmatic user interface, they may specify the procedures to be replaced/enhanced and the procedures that will actually be executed in their place, or in addition to them. These procedure associations are then stored by the Architected Interface Facility: Procedure Exits product and used at run time to alter the affected calling sequences for one or more processes.

The Architected Interface Facility: Procedure Exits product introduces several new concepts into the MPE world. For example, the actual execution of certain procedures in place of, or in addition to, other procedures is referred to in this document as interception. Consequently, a prerequisite to understanding the Architected Interface Facility: Procedure Exits product, and subsequently its use, is a good understanding of the key concepts and terminology used in its description.

Key Concepts and Terminology

This section explains the several key concepts introduced by the Architected Interface Facility: Procedure Exits product and, in doing so, defines much of the terminology used to describe the product.

Interception

The Architected Interface Facility: Procedure Exits product allows users to **intercept** certain procedures with other procedures. This simply means that some procedure(s) execute in place of, or in addition to, other procedures. The routine being intercepted is called the **target procedure**, and the routine that is executed upon interception is called the **handler procedure**.

Handler procedures can either be executed before or after the target procedure. Those executed before are referred to as **invocation handlers**; those executed after are **termination handlers**.

Invocation handlers may determine at run time whether or not to **stub out**, or skip, execution of the target procedure. If an invocation handler stubs out the target procedure, then all invocation handlers and termination handlers of lesser priority armed on the same target will also be skipped.

Binding

With the Architected Interface Facility: Procedure Exits product, users specify what handlers to execute for which targets. Each of these handler/target associations is called a **binding**. The act of creating a binding is called an **arming**. Thus, a user would arm a handler on a target to create a binding.

Once a target is armed, any subsequent calls to that target are intercepted, and the handler procedures are invoked. Conversely, when a target is disarmed, then the target procedure is no longer intercepted, and the handler procedures are no longer executed. Arming and disarming can take place at any time on a currently running system, and no reboot or relink is necessary. The effects are immediate.

Handler Priority

All handlers are ordered by **priority**, which is an integer value assigned to the **library** in which the handler resides. This means that all handlers in the same library have the same priority. Invocation handlers are executed in order of decreasing priority, and termination handlers are executed in order of increasing priority. The invocation handler with the highest priority is the first to be executed; the termination handler with the highest priority is the last to be executed.

Handler priorities not only provide users a way to order multiple handlers on the same target, but they are also used to prevent mutual recursion in handlers (infinite looping).

Scope of Bindings

PEARM allows you to define a binding for any existing process on the system. You may specify that bindings are to be bequeathed (inherited) to any of its child process(es), existing or future. Thus, a binding can be in effect for an entire process tree.

You can define a binding that will be in effect for every process (existing or future) on the system by calling PEARM with PIN=1 and BEQUEATHE=TRUE.

The scope of a binding is **process local** if the binding is defined for the caller of PEARM only. If the binding is bequeathed or if it is defined for another process, then the scope is **process external.**

User-Defined System Libraries

MPE/iX supports two NM system libraries of executable routines; these are NL.PUB.SYS and XL.PUB.SYS. These libraries provide run time support for applications and operating system software without the need for users to specify them explicitly when running a program (for example, with the RUN command). 'NL.PUB.SYS" is the last library to be searched for any given external reference, and it may not make any external calls of its own. XL.PUB.SYS is slightly different in that its external references may be resolved through NL.PUB.SYS, and it is the next to last library to be searched for external references within the program file. It is important to note, however, that some subsystems, like the Command Interpreter, do not load XL.PUB.SYS.

By definition, system handlers may only be resolved through system libraries, since only these libraries are loaded for all programs. This allows the Architected Interface Facility: Procedure Exits product to find the routine when it needs to be executed. Rather than require all system handler routines to be placed in XL.PUB.SYS the Architected Interface Facility: Procedure Exits product introduces the concept of **User Defined System Libraries (UDSLs).**

A UDSL is defined as a library that must be able to have its externals resolved through only NL.PUB.SYS. This allows the Architected Interface Facility: Procedure Exits product to dynamically load a library during execution, so that handler routines may be referenced by any program. Remember, since not all processes load XL.PUB.SYS, the definition is restricted to NL.PUB.SYS only.

A library is considered to be a UDSL if a handler within the library is bound with a process external scope. The Architected Interface Facility: Procedure Exits product verifies that the library is not already loaded so that Architected Interface Facility: Procedure Exits can ensure that the library is only resolved through NL.PUB.SYS. The library then remains open until the handler is disarmed and all process external processes that have loaded this library terminate.

Note

During an interception, Architected Interface Facility: Procedure Exits will dynamically load the UDSL library where the handler resides. This means that the process for which this interception takes place will have the library loaded. The library will not be unloaded until this process terminates. This point is also explained in the README.PE.SYS file, Chapter:6 Common Questions and Answers.

Design Considerations

This section presents in more detail the design considerations of the Architected Interface Facility: Procedure Exits product and its internal mechanisms.

Interception

The key feature of the Architected Interface Facility: Procedure Exits product is its ability to intercept calls to one procedure, and then execute others in addition to it, or in its place. This requires the ability to catch any call that is made to the target procedure. Within MPE/iX, there are basically two types of procedure calls, internal and external. In the following paragraphs, both types of calls are discussed, but the Architected Interface Facility: Procedure Exits product actually only intercepts external calls. If internal calls need to be intercepted, then the procedures to be intercepted must be **instrumented**, or linked, in a way that forces internal calls to external calls.

Caution

Hewlett-Packard support for internal procedures, both instrumented and non-instrumented, extends only to the Architected Interface Facility: Procedure Exits software. Hewlett-Packard does not guarantee nor provide support for the internal procedures. An internal procedure may function differently from release to release and may even disappear over time. To ensure that your application software is not adversely affected by such changes to internal procedures, it is recommended that you thoroughly test your application with each release of the MPE/iX operating system. If you require access to a pre-release version of the operating system for this testing, please contact your Hewlett-Packard support representative.

External Calls

External calls may also be referred to as **Inter-SOM** calls. A System Object Module (SOM) is the smallest loadable unit of an executable library and contains the object code for a number of procedures. Each SOM has its own space and a unique data pointer (DP) and link pointer (LP). Calling a procedure in another SOM requires the saving off of the current DP and LP, and the loading of the new DP and LP. This is done via **cross reference table (XRT) entries**, **import stubs**, **export stubs**, and `callx` routines.

At load time, each external procedure called by a SOM is allocated an XRT entry in the process's local space. This XRT entry contains the address of a callx routine that will read the rest of the XRT entry and perform the external call. Import stubs and export stubs are provided by the linker for entering and exiting different SOMs. Thus, an external procedure call involves a lookup of the XRT entry for the called procedure and a branch to the callx using an import stub, and then a branch back to the caller using an export stub.

The Architected Interface Facility: Procedure Exits product alters this sequence in order to intercept a target procedure. When a target is armed with a handler to create a binding, Architected Interface Facility: Procedure Exits changes the callx address of the target procedure within all the XRT entries of all the processes within the scope of the binding to the address of the corresponding Architected Interface Facility: Procedure Exits product callx. The Architected Interface Facility: Procedure Exits callx then transfers control to an Architected Interface Facility: Procedure Exits procedure that causes the handler routines to be invoked in the appropriate order.

Internal Calls

Internal procedure calls are also known as **Intra-SOM** calls, or procedure calls made between routines in the same SOM. Unlike external calls, internal calls do not use XRTs, import stubs, export stubs, or a callx routine. They are resolved at link time, and consist only of a simple branch and link instruction.

In order for Architected Interface Facility: Procedure Exits to intercept internal calls, the calls must be made into external calls. For MPE/iX operating system routines and HP subsystem routines, a special relinking of the target procedure may be done to ensure all calls to it are made external. For user procedures to be intercepted, this requires that target procedures reside in separate SOMs to ensure that any calls are made in an external fashion.

Interception from Compatibility Mode

For users executing compatibility mode (CM) programs, the Architected Interface Facility: Procedure Exits product provides a separate means for interception. This requires some effort by the user in that the procedures to be intercepted must be **instrumented**, or modified, to call Architected Interface Facility: Procedure Exits. Architected Interface Facility: Procedure Exits product includes `PEINST` for user instrumentation of CM procedures.

A user instruments a procedure by first creating a native mode (NM) calling procedure. This procedure may have any name the user chooses, since it must be specified at arming time. The user must then modify the CM procedure to be intercepted by placing as the first statement a call to the procedure exits AIF PEINST. PEINST expects as a parameter the plabel, or procedure address, of the NM procedure. Procedure exits AIFs can then execute the handlers that the user has armed onto the procedure. After all handlers have been executed, procedure exits AIFs return control to the CM procedure.

Handler Prioritization

The Architected Interface Facility: Procedure Exits product utilizes a priority schema for ordering the execution of handlers bound to the same target procedure. This accomplishes two specific purposes. It first allows users to decide how they want their own handlers ordered, with some control over the order relative to other users' handlers. Secondly, it prevents mutual recursion among handlers. Both of these are explained further in the following paragraphs.

Prioritizing Multiple Handlers

The priority schema itself is based on the libraries in which the handler and target procedures reside. Each library, which is specified at arming time, is assigned an integer priority between one (1) and MAXINT-1, with one (1) being the lowest priority and MAXINT-1 being the highest priority. Priorities one (1) and two (2) are reserved for NL.PUB.SYS and XL.PUB.SYS, respectively. This means that invocation handlers residing in XL.PUB.SYS are always last to be executed, and termination handlers residing in XL.PUB.SYS are always the first to be executed. It is important to note that, in order to ensure proper prioritization of handlers, there can be at most one handler for a particular target residing in the same library.

Users may either request an explicit priority for a particular library, or have the Architected Interface Facility: Procedure Exits product assign one for them. Procedure exits AIFs always assign the lowest priority available. For example, if an invocation handler must be the first to be invoked, then the user can request a priority of MAXINT-1 for the library in which the procedure resides. If MAXINT-1 is already taken, then the arming will fail, and the user can then decide if he/she wishes to assign a lower priority. Priorities are recycled when the library is no longer needed (no procedures from the library are bound either as targets or handlers).

Preventing Mutual Recursion in Handlers

The act of interception introduces the possibility of **mutual recursion** among handlers. Consider the following scenario:

1. Target (T1) armed with Handler (H1)
2. Target (T2) armed with Handler (H2)
3. Handler (H1) calls Target (T2)
4. Handler (H2) calls Target (T1)

Without any checks or priority schema, this would lead to an execution sequence of:

1. Target (T1) is intercepted by Handler (H1)
2. Handler (H1) calls Target (T2)
3. Target (T2) is intercepted by Handler (H2)
4. Handler (H2) calls Target (T1)
5. Go to #1—> Infinite loop!

Without the Architected Interface Facility: Procedure Exits priority schema, the user would not be able to prevent this infinite looping, since handlers may be installed by two different products, or handler calls to target procedures may be indirect and the calling sequence would be essentially hidden (for example, Handler (H2) calls Procedure (A) which calls Target (T1)).

The Architected Interface Facility: Procedure Exits product prevents mutual recursion through the conjunction of unique handler and/or target priorities and a process local Architected Interface Facility: Procedure Exits priority stack, whose top is the priority of the last launched handler that is currently executing. The top of the stack is referred to as the **current priority**. If the stack is empty, then the default current priority is MAXINT. Before a handler is executed, its priority is compared to the current priority and the target priority. The following rule is checked: Is **Target Priority < Handler Priority < Current Priority** ?

If this case is true, then the handler priority is pushed onto the priority stack and becomes the current priority; it is popped off the stack when it returns to its caller, a procedure exits AIF. To illustrate how this prevents mutual recursion, take another look at the previously mentioned example, adding priorities and using the priority stack mechanism. Let the priorities for the targets and handlers be as follows:

- Target (T1) Priority = 1
- Target (T2) Priority = 2
- Handler (H1) Priority = 3
- Handler (H2) Priority = 4

Now the order of execution becomes the following:

1. Target (T1) is intercepted by Handler (H1)
2. Perform Check
   a. Target (T1) Pri = 1 < Handler (H1) Pri = 3 < Current Pri = MAXINT ?
   b. Check Passed!
3. Handler (H1) Pri (3) Pushed onto Priority Stack
4. Handler (H1) calls Target (T2)
5. Target (T2) is intercepted by Handler (H2)
6. Perform Check
   a. Target (T2) = 2 < Handler (H2) Pri = 4 < Current Pri = 3 ?
   b. Check Failed! Handler (H2) not Executed!
7. Execute Target (T2)
8. Return to Handler (H1)
9. Return to Target (T1)
10. Return to Caller

In the above example, the recursion is prevented with the Architected Interface Facility: Procedure Exits priority methods employed. It is important to note that Handler (H2) never gets executed, and thus Target (T2) is not intercepted since it is called by a higher priority handler.

Note

This priority schema prevents a handler from being bound to a target in the same library. Therefore, users should ensure that handlers to be armed on a particular target reside in a separate library from the target itself.

3

Procedure Exits Architected Interfaces

This chapter describes the procedure exits architected interfaces (AIFs). Detailed descriptions of procedure exits AIFs are located in the following chapter. In addition, the interactive utility PEUTIL is also described.

Types of Architected Interfaces

There are four types of procedure exits AIFs:

- Access management AIFs
- Exit arming AIFs
- CM instrumentation AIF
- Utility AIFs

Access Management Architected Interfaces

Access management AIFs are:

- `AIFACCESSON`
- `AIFACCESSOFF`

Access management AIFs provide a mechanism, the user ID, to validate user access to procedure exits AIFs.

User IDs

Each purchaser of the Architected Interface Facility: Procedure Exits product is assigned a unique user ID. Whenever you call an AIF, you must identify yourself by using your company's user ID. The AIF validates the user ID against a table stored in the procedure exits AIF's internal data area.

Each AIF includes an optional *user_id* parameter. If your program is only going to make a small number of AIF calls, you'll want to pass the user ID to each AIF as you call it.

However, if your program is going to make a lot of AIF calls, there is a more efficient method to specify your user ID. If your application uses the `AIFACCESSON` AIF to pass your user ID, all subsequent AIF calls made by your application will be assumed to belong to the same user ID. Use `AIFACCESSOFF` after completing the multiple AIF calls.

Note

You must use the user ID installed through AIFGLOBINSTALL in all calls to AIFs described in this manual. If you have purchased another Architected Interface Facility product (for example, operating system AIFs), you still need to call AIFACCESSON with the Architected Interface Facility: Procedure Exits user ID in order to call procedure exits AIFs, even if you have called AIFACCESSON for the other product.

What Is the Purpose of User IDs?

Architected Interface Facility user IDs are used by Hewlett-Packard Response Centers to ensure that AIF-based software products are properly supported. The user IDs are not intended to prevent users who have not purchased an Architected Interface Facility product from calling AIFs; instead, user IDs are intended to guarantee the best possible support.

Because AIFs are trusted procedures, their misuse can cause a number of system problems (including system failures and data corruption). If this should happen, Hewlett-Packard's Response Centers can determine the user IDs associated with any AIF calls that result in errors. In this way, identifying and fixing AIF-related system problems can be accomplished quickly.

Exit Arming Architected Interfaces

Exit arming architected interfaces are:

- PEARM
- PEDISARM

The Architected Interface Facility: Procedure Exits exit arming AIFs allow the user to arm target procedures with handlers.

CM Instrumentation Architected Interface

The CM instrumentation AIF is PEINST. CM instrumentation involves first creating a native mode (NM) calling stub for the procedure to be intercepted. This stub is simply a dummy procedure to which the user will bind any handlers for the CM target. It may have any name the user chooses, and this name must be specified when making calls to PEARM to specify bindings.

The second step involves actually modifying the source of the CM target to make a call to PEINST (located in the CM system library SL.PUB.SYS). The call to PEINST must be the first statement in the procedure.

In order to access the parameters of the CM target from the handler, which must be in NM, it may be necessary to convert CM relative addresses into NM virtual addresses. Use the utility AIF, AIFCONVADDR, to perform this task.

Utility Architected Interfaces

Utility AIFs provide miscellaneous functionality useful to application developers.

- AIFGLOBINSTALL programmatically installs the Architected Interface Facility: Procedure Exits user ID onto a system.
- AIFCONVADDR converts compatibility mode relative addresses to the corresponding native mode virtual address.

Allowing and Disallowing Procedure Exits

Even though the Architected Interface Facility: Procedure Exits product is primarily a software developer product, end customers may require the ability to ensure at any point in time that the Architected Interface Facility: Procedure Exits product is not enabled on a particular system. PEUTIL is provided in order to allow users to accomplish this task.

Starting PEUTIL

PEUTIL is executed by using the RUN command as follows:

```
:RUN PEUTIL
peutil>
```

Allowing Procedure Exits

The syntax for using the allow command is:

```
allow [;susp=true/false]
```

This allows procedure exits use on the system. If susp=true is specified, it uses a suspended version of procedure exits, else it discards all older bindings and reinitializes the product. By default, susp is true.

Disallowing Procedure Exits

The syntax for using the disallow command is:

```
disallow [;susp=true/false]
```

This disallows procedure exits on the system. Usage of the AIFs, PEARM, PEDISARM, PELOCK, and PEUNLOCK, as well as execution of any handlers, is prevented. If susp=true is specified, it suspends the current version of the product, saving all current bindings. Otherwise, all current bindings will be deleted and the product will be reinitialized. By default, susp is true.

Exiting PEUTIL PEUTIL may be terminated by using the command EXIT as follows:

```
peutil>exit
```

4

Architected Interface Programming Considerations

This chapter describes the rules for architected interface (AIF) use, including:

- Data types used to declare AIF parameters.
- Mappings of the data types to programming languages that support AIF use.
- The AIF error management strategy.

Data Type Naming Convention

Below are listed the generic data types (and their mnemonics) that are used to declare the types for the AIF parameters and the values returned.

Table 4-1. Types Used and Their Mnemonics

Mnemonic	Generic Data Type
I32	32-bit signed integer.
U32	32-bit unsigned integer.
B	Boolean.
C	Character.
@32	32-bit address.
@64	64-bit address.
A	Array. Used in combination with other types. For example, CA represents an array of elements, each element containing an ASCII character value; BA represents an array of elements, each element containing a boolean value.
Rec	Record. Refer to appendix B for record structures.

Data Type Mappings to Languages

Most of the information exchange across the AIF interfaces is accomplished through the use of scalar types, which do not require any special treatment. The scalar types include integers, short integers, character arrays, and booleans.

For record types, the documentation provides the Pascal record declaration as well as the packing of the fields as implemented on the HP Pascal/iX compiler. This information should make the call usable from Pascal and C.

Refer to the following manuals for further information on HP Pascal/iX:

HP Pascal Reference Manual (31502-90001)
HP Pascal Programmer's Guide (31502-90002)

Refer to the following manuals for further information on HP C/iX:

HP C/iX Reference Manual (31506-90005)
HP C Programmer's Guide (92434-90002)

Note

If the C programming language is used, all AIF names must be specified in upper case.

Error Management

Error checking has been kept to a minimum for increased performance. AIFs use the parameter *overall_status* to indicate the status of the call, on the whole. The data type is `status_type`:

```
status_type = record
              case boolean of
                   true : (all    : integer);
                   false: (info   : shortint;
                           subsys: shortint);
              end;
```

If an AIF detects no errors, it returns a 32-bit integer with a value of zero. If errors are detected, returns an array of two 16-bit integers. The leftmost 16 bits contain the actual error number, and the rightmost 16 bits contain the subsystem number.

The AIF subsystem number is 516, so AIF errors are reported with a subsystem number of 516. In some cases, the AIFs call another subsystem; if that subsystem detects an error, the called subsystem's number may be returned instead. The information halfword contains the error message number. Appendix A has a complete list of the AIF error messages for the AIF:PE product. A zero return indicates normal execution. A positive number indicates a warning. A negative number indicates an error condition for the overall call.

5

Architected Interface Descriptions

This chapter describes procedure exits architected interfaces, arranged alphabetically.

AIFACCESSOFF

Deactivates the user ID for the current process.

Syntax

```
                        REC         I32
AIFACCESSOFF (overall_status, user_id)
```

Parameters

overall_status — **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: `status_type` (Refer to appendix B.)

user_id — **32-bit signed integer by value (required)**

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Procedure Exits product.

Operation Notes

None.

AIFACCESSON

Validates the user ID and authorizes the calling process to call AIFs.

Syntax

```
                    REC          I32
AIFACCESSON (overall_status, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: `status_type` (Refer to appendix B.)

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Procedure Exits product.

Operation Notes

Each process that makes AIF calls must provide a valid Architected Interface Facility: Procedure Exits user ID. Once this procedure is called, the process remains authorized until it terminates or until it calls `AIFACCESSOFF` to deactivate the user ID. Any subsequent AIF calls do not require a *user_id* parameter for validation.
If `AIFACCESSON` is not called, the user ID must be passed as a parameter with every AIF call.

AIFCONVADDR

Converts compatibility mode relative addresses to the corresponding native mode virtual addresses.

Syntax

```
                    REC            I32A            RECA
AIFCONVADDR (overall_status, mode_array, inaddress_array,
                    @64A            I32A            I32
             outaddress_array, convstatus_array, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: `status_type` (Refer to appendix B.)

mode_array **32-bit signed integer array by reference (required)**

Passes an array of integers where each element contains a value indicating the addressing mode of the CM address located in the corresponding element in ***inaddress_array***. If n addresses are being converted, element $n + 1$ must be a zero to indicate the end of the element list.

Values and their meanings are as follows:

1	DB relative byte address (stack or XDS)
2	DB relative word address (stack or XDS)
3	Stack DB relative byte address (stack only)
4	Stack DB relative word address (stack only)
5	Bank 0 relative word address

inaddress_array **record array by reference (required)**

An array where each element is an unsigned 16-bit CM address to be converted. The addressing mode of the CM address is defined in the corresponding element in ***mode_array***.

Array type: `bit16` (Refer to appendix B.)

outad-dress_array **64-bit address array by reference (required)**

An array where each element returns a 64-bit address that is the result of the conversion performed on a CM address located in the corresponding element in ***inaddress_array***.

Array type: `globalanyptr`

convstatus_array **record array by reference (required)**

An array where each element returns the status of the conversion operation performed in the corresponding element in ***inaddress_array***. A zero indicates a successful operation. A negative value indicates an error condition. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Array type: `status_type` (Refer to appendix B.)

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Operating System product. If it is not passed, the caller must have previously called `AIFACCESSON`, or the call fails.

Default: 0

Operation Notes

None.

AIFGLOBINSTALL

Installs the user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Procedure Exits product. AIFGLOBINSTALL enables an application to execute procedure exits AIF code located on the target 900 Series HP 3000 computer system.

Syntax

```
                        REC          I32
AIFGLOBINSTALL (overall_status, user_id)
```

Parameters

overall_status — **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: status_type (Refer to appendix B.)

user_id — **32-bit signed integer by value (required)**

Passes the user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Procedure Exits product.

Operation Notes

AIFGLOBINSTALL must be executed on all systems containing code that calls procedure exits AIFs (for example, your application). It should be executed once per installation. However, it can be executed each time your application is run without side effects. Your application must execute AIFGLOBINSTALL prior to calling any other procedure exits AIFs.

AIFGLOBINSTALL will fail if not enough disk space is located on LDEV 1. If this occurs, you must create additional free space on LDEV 1 before attempting to re-execute code that contains the call to AIFGLOBINSTALL.

PEARM

Arms a handler routine onto a target routine. PEARM specifies the characteristics of the binding being requested, including target and handler priority, type of the handler, and the scope of the binding.

Syntax

```
                REC             REC             REC          REC          REC
PEARM (overall_status, handler_proc, target_proc, handler_lib, target_lib,
              I32         I32          I32          I32          REC          I32
       handler_pri, target_pri, handler_type, bind_id, prod_name, pin,
                B       B     I32
       bequeathe, cm, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: `status_type` (Refer to appendix B.)

handler_proc **record by reference (required)**

Name of the handler procedure that is to be armed onto a target procedure. First character is assumed to be a delimiter for start and end of character array. Name is case sensitive.

Record type: `aifpe_procname_type` (Refer to appendix B.)

target_proc **record by reference (required)**

Name of the target procedure that is being armed with a handler procedure. First character is assumed to be a delimiter for start and end of character array. Name is case sensitive.

Record type: `aifpe_procname_type` (Refer to appendix B.)

handler_lib **record by reference (optional)**

Name of library in which handler procedure can be found. First character is assumed to be a delimiter for start and end of character array.

Group and account are defaulted appropriately, as in the HPGETPROCPLABEL intrinsic.

If the binding is defined for a process other than the calling process or if bequeathe is true, this library is considered a User Defined System Library (UDSL). As such, it must be loadable only through NL.PUB.SYS and may not already be open.

Record type: aifpe_filename_type (Refer to appendix B.)

Default: XL.PUB.SYS

target_lib **record by reference (optional)**

Name of library in which target procedure can be found. First character is assumed to be a delimiter for start and end of character array. Group and account are defaulted appropriately, as in the HPGETPROCPLABEL intrinsic.

If the binding is defined for a process other than the calling process or if bequeathe is true, this library is considered a User Defined System Library (UDSL). As such, it must be loadable only through NL.PUB.SYS and may not already be open.

Record type: aifpe_filename_type (Refer to appendix B.)

Default: NL.PUB.SYS

handler_pri **32-bit signed integer by reference (optional)**

Priority to be given to handlers residing in *handler_lib*. If user-specified, Architected Interface Facility: Procedure Exits attempts to assign requested priority; if taken, PEARM returns an error status. If not specified, Architected Interface Facility: Procedure Exits will assign the lowest priority available.

A value of -1 explicitly requests Architected Interface Facility: Procedure Exits to assign the lowest available priority. If the parameter is passed, PEARM returns the assigned priority via this parameter.

Invocation handlers are executed in order of decreasing priority; conversely, termination handlers are executed in order of increasing priority. Priority one (1) is reserved for NL.PUB.SYS; priority two (2) is reserved for XL.PUB.SYS. Requesting MAXINT-1 will ensure that a library's handlers will be executed first upon invocation and last upon exit. However, there cannot be more than one handler from the same library bound to the same target.

Default: Lowest available priority

target_pri **32-bit signed integer by reference (optional)**

Priority to be given to targets residing in *target_lib*. If user-specified, Architected Interface Facility: Procedure Exits attempts to assign the requested priority; if it has already been assigned, PEARM returns an error status. If not specified, Architected Interface Facility: Procedure Exits assigns the lowest priority available.

A value of -1 explicitly requests Architected Interface Facility: Procedure Exits to assign the lowest available priority. If the parameter is passed, PEARM returns the assigned priority via this parameter.

Target procedures may only be intercepted by handler procedures of higher priority. This means that a target may not be bound to a handler in the same library.

Default: Lowest available priority

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

Passes the type of handler procedure; either invocation or termination. A value of 0 means that the handler should be called upon invocation of target. A value of 1 indicates that it should be called upon termination of target.

Default: 0

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

Passes a user-specified binding ID. It is unique to a target/handler pair and will be passed to the handler as part of the calling sequence.

Default: 0

prod_name **record by reference (optional)**

Passes the name of the product or software developer to be associated with this binding.

Record type: `aifpe_prodname_type` (Refer to appendix B.)

Default: Blanks

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

Passes the pin of the process for which the binding is to be defined. The default is the calling process. This parameter is used in conjunction with the *bequeathe* parameter to define the scope of the binding. If the *bequeathe* parameter is true, the binding will be in effect for this pin and all of its children, existing or future. If the *bequeathe* parameter is false, the binding will be in effect for this process only.

Default: Pin of the calling process

bequeathe **boolean by value (optional)**

Passes a boolean value denoting whether this binding is to be bequeathed to any of the descendants of the root process specified by the *pin* parameter. This parameter is used in conjunction with the *pin* parameter to define the scope of the binding.

Default: False

cm **boolean by value (optional)**

Passes a boolean value denoting whether this target procedure is to be called from CM or from NM. A value of True indicates CM; a value of False indicates NM.

Default: False (NM)

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Operating System product. If it is not passed, the caller must have previously called `AIFACCESSON`, or the call fails.

Default: 0

Operation Notes

None.

PEDISARM

Disarms a handler routine from a target routine. PEDISARM specifies the characteristics of the binding to be deleted.

Syntax

```
                REC             REC          REC          REC        REC
PEDISARM (overall_status, handler_proc, target_proc, handler_lib, target_lib,
                I32      I32    B        I32
          handler_type, pin, bequeathe, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: `status_type` (Refer to appendix B.)

handler_proc **record by reference (required)**

Name of the handler procedure that is to be disarmed from a target procedure. First character is assumed to be a delimiter for start and end of character array. Name is case sensitive.

Record type: `aifpe_procname_type` (Refer to appendix B.)

target_proc **record by reference (required)**

Name of the target procedure that is being disarmed. First character is assumed to be a delimiter for start and end of character array. Name is case sensitive

Record type: `aifpe_procname_type` (Refer to appendix B.)

handler_lib **record by reference (optional)**

Name of library in which handler procedure resides. First character is assumed to be a delimiter for start and end of character array.

Record type: `aifpe_filename_type` (Refer to appendix B.)

Default: Nil

target_lib **record by reference (optional)**

Name of library in which target procedure resides. First character is assumed to be a delimiter for start and end of character array.

Record type: `aifpe_filename_type` (Refer to appendix B.)

Default: Nil

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

Type of handler procedure being disarmed from the target procedure. A value of 0 indicates an invocation handler; a value of 1 indicates a termination handler.

Default: 0

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

Passes the pin of the process for which the binding is to be removed. The default is the calling process. This parameter is used in conjunction with the *bequeathe* parameter to define the scope of the binding. If the *bequeathe* parameter is true, the binding will be removed for this pin and all of its children, existing or future. If the *bequeathe* parameter is false, the binding will be removed for this process only.

Default: Pin of the calling process

bequeathe **boolean by value (optional)**

Passes a boolean value denoting whether disarming is to affect any of the descendants of the root process specified by the *pin* parameter. This parameter is used in conjunction with the *pin* parameter to define the scope of the disarming.

Default: False

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Operating System product. If it is not passed, the caller must have previously called `AIFACCESSON`, or the call fails.

Default: 0

Operation Notes

None.

PEINST

CM callable only.

Used to intercept CM target procedures residing in user code.

Syntax

```
        U32
PEINST (plabel)
```

Parameters

plabel **32-bit unsigned integer by value (required)**

The procedure label (in HPLOADNMPROC format) of the NM calling stub to which the user binds any handlers for the CM target procedure.

Operation Notes

A call to PEINST must appear as the first statement of the CM target procedure. Refer to chapter 2 for more information about interception of CM procedures

PELOCK

No longer functional as of MPE/iX release 5.0.

The Architected Interface Facility: Procedures Exit product used to provide a way to obtain exclusive access to Procedure Exits data and functions. Due to locking strategy issues, this functionality has been removed in MPE/iX release 5.0. PELOCK and PEUNLOCK have been left for compatibility reasons, but do not provide any functionality.

Syntax

```
                REC         I32
PELOCK (overall_status, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: status_type (Refer to appendix B.)

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Operating System product. If it is not passed, the caller must have previously called AIFACCESSON, or the call fails.

Default: 0

Operation Notes

As of MPE/iX release 5.0, PELOCK just returns an ***overall-status*** of zero.

PEUNLOCK

No longer functional as of MPE/iX release 5.0.

The Architected Interface Facility: Procedures Exit product used to provide a way to obtain exclusive access to Procedure Exits data and functions. Due to locking strategy issues, this functionality has been removed in MPE/iX release 5.0. PELOCK and PEUNLOCK have been left for compatibility reasons, but do not provide any functionality.

Syntax

```
                REC         I32
PEUNLOCK (overall_status, user_id)
```

Parameters

overall_status **record by reference (required)**

Returns the overall status of the call. A zero indicates a successful call. A negative value indicates an error in the overall call. A positive value indicates a warning. Refer to appendix A for meanings of status values.

Record type: status_type (Refer to appendix B.)

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

The user ID assigned to a vendor at the time of purchase of the Architected Interface Facility: Operating System product. If it is not passed, the caller must have previously called AIFACCESSON, or the call fails.

Default: 0

Operation Notes

As of MPE/iX release 5.0, PEUNLOCK just returns an ***overall-status*** of zero.

6

Architected Interface Examples

Description of Example Programs

This chapter provides programming examples of using the procedure exits architected interfaces.

There are three basic steps in using the procedure exits architected interfaces:

1. Create Handler
2. Arm Handler
3. Disarm Handler

The following three programs correspond with these three steps. The three example programs are shown first written in PASCAL and then written in C.

PASCAL Examples

The purpose of PASCAL handler is to disallow file equations. The handler will accomplish this by **intercepting** calls to the NL procedure fopen_nm(). After the handler is **armed**, processes that try to open a file will be **intercepted** by the handler.

When the handler is passed a file equation (for example, "*x"), the handler will **stubout** (skip) the call to fopen_nm(). If the handler is not passed a file equation, the handler will allow the call to fopen_nm(), to open the file.

The PASCAL handler procedure should be executed before the target procedure. Handlers of this type are called **invocation handlers**. Invocation handlers may determine at run time whether or not to stub out, or skip, the execution of the target procedure. In the PASCAL example, the handler will stubout the target procedure when a file equation is passed.

The Arm Handler program **arms** the handler procedure "user_handler". The **target procedure** is fopen_nm(), which is an internal procedure in the NL. The handler/target association is called a **binding**. The Arm Handler program arms the binding.

The Disarm Handler program disarms the handler procedure "user_handler" from the target procedure fopen_nm(). The binding between the handler and the target is deleted.

C Examples

The purpose of C handler is to intercept calls to HPFOPEN. The handler will display the HPFOPEN arguments that have been passed by the caller.

After the handler is **armed**, processes that call HPFOPEN will be **intercepted** by the handler. The handler will display the HPFOPEN arguments that have been passed by the caller. After the handler has displayed the HPFOPEN arguments it will transfer control to HPFOPEN.

The C handler procedure should be executed before the target procedure. Handlers of this type are called **invocation handlers**.

This Arm Handler program **arms** the handler procedure hpfopen_handler(). The **target procedure** is HPFOPEN. The handler/target association is called a **binding**. The Arm Handler program arms the binding.

The Disarm Handler program disarms the handler procedure hpfopen_handler() from the target procedure HPFOPEN. The binding between the handler and the target is deleted.

Handler (PASCAL)

```
{------------------------------------------------------------------------}
{ This program intercepts the target fopen_nm(). The goal is to disallow }
{ the use of file equations. If a file equation is passed (say "*x"), we }
{ stubout (skip) the call to fopen_nm().  If no file equation is passed  }
{ we let the open of the file occur, that is, we don't stubout the target}
{ Since this handler must execute before the target, fopen_nm(), is      }
{ called, it's an INVOCATION_HANDLER.                                    }
{                                                                        }
{ DISCLAIMER: This pgm is for illustration only. It has no real life use.}
{                                                                        }
{ To compile -> :pasxl thisFile                                          }
{                                                                        }
{ To link    -> :linkedit                                                }
{               LinkEd> buildxl testxl.pe; limit=1                       }
{               LinkEd> addxl $oldpass; privlev=2                        }
{               LinkEd> exit                                             }
{                                                                        }
{ To use     -> Arm this handler using the arming pgm provided. You may  }
{               want to create a CI, son of your current CI. Identify    }
{               its Pin# with the :showproc command. Pass this Pin# to   }
{               the arming pgm.                                          }
{               Once the arming is done, issue a file equation and try   }
{               to :print the file with/without the file equation.       }
{------------------------------------------------------------------------}
$standard_level 'Ext_Modcal'$
$subprogram$
```

```
program User_Handler_to_intercept_fopen_nm;
type
  designator_type    = packed array [1..1024] of char;

  fopen_nm_args_type = record
                       { Note that the arguments are }
                       { declared in reverse order.  }
                       filecode       : shortint;
                       initalloc      : shortint;
                       numextents     : shortint;
                       filesize       : integer;
                       numbuffers     : shortint;
                       blockfactor    : shortint;
                       userlabels     : shortint;
                       formmsg        : localanyptr;
                       device         : ^designator_type;
                       recsize        : shortint;
                       aoptions       : shortint;
                       foptions       : shortint;
                       formdesignator : ^designator_type;
                       end;


procedure HPSetccode; intrinsic;
procedure Print;      intrinsic;

$sysintr 'aifintr.pub.sys'$
procedure AifProcPut; intrinsic;

procedure user_handler(    bindid        : integer;
                       var stubout       : boolean;
                           parms_area    : localanyptr;
                       var func_rtn_area : shortint ) option extensible 4;
const
  AIF_MY_VENDOR_ID   = 1234;
  CCL                = 1;
  SECURITY_VIOLATION = 93;

var
  feq_not_allowed_msg: packed array [1..32] of char;
  fopen_nm_arg_ptr   : ^fopen_nm_args_type;
  fserr              : integer;
  item               : globalanyptr;
  itemnum_array      : array [1..2] of integer;
  itemstatus         : integer;
  overall_status     : integer;
```

```
begin
  { initialize the pointer to the argument list passed by the caller to }
  { fopen_nm(). Since the number of arguments passed to the target is   }
  { unknown to AIF:PE, the handler is called with the ptr "parms_area"  }
  { pointing to the first word on the stack after the argument list.    }
  { The handler just has to substract the length of the argument record }
  { from "parms_area" to get the ptr to the whole argument list passed. }
  {                                                                     }
  {      /            /                                                 }
  {      |  caller()  |        arg0..argN is the list of args that can  }
  {      |            |  (2)   be passed by the caller to the target.   }
  {      +------------+ <--+                                           }
  {      |   arg N    |    |   "filecode"        ! In the handler, the  }
  {      +------------+    |                     ! declaration of the   }
  {      |            |    |        o            ! args passed must     }
  {      /            /    |        o            ! match the stack      }
  {      |            |    |        o            ! layout, that is,     }
  {      +------------+    |                     ! we have to declare   }
  {  (1) |   arg 0    |    |   "formdesignator"  ! the args in the      }
  { +--> +------------+    |                     ! reverse order.       }
  { |    |Frame Marker|    |                                            }
  { |    +------------+    |   (1): "parms_area" points to the first    }
  { |    |   AIF:PE   |    |        word after arg0 since the number    }
  { +----|  internal  |    |        of args passed is unknown to AIF:PE }
  {      |  routines  |    |                                            }
  {      +------------+    |   (2): Handler knows the number and size   }
  {      |Frame Marker|    |        of the args expected by the target. }
  {      +------------+    |        It can use the ptr "parms area" to  }
  {      |user_       |    |        build the ptr "fopen_nm_args_ptr"   }
  {      |   handler()|----+        that will point to the whole list   }
  {      +------------+             of args passed by the caller of     }
  {      |Frame Marker|             fopen_nm().                         }
  {      +------------+                                                 }

  fopen_nm_args_ptr := addtopointer( parms_area, -sizeof(fopen_nm_args_type) );
```

```
    { check if first character is a "*" }
    if (fopen_nm_arg_ptr^.formdesignator^[1] = '*') then
      begin
      { inform user that we do not accept file equations. }
      no_feq_msg := #27'&dC No file equations allowed! ';
      Print( no_feq_msg, -sizeof(no_feq_msg), 0 );

      { refuse to open the file by skipping the call to fopen_nm() }
      stubout := TRUE;

      { return 0, that is, failure to open file, to the caller of fopen_nm() }
      func_rtn_area := 0;

      { set the ccode to CCL to indicate failure. Hope someone will check it }
      HPSetccode( CCL );

      {call AifProcPut() to set the FSerr to Security Violation}
      itemnum_array[1] := 2075;
      fserr            := SECURITY_VIOLATION;
      item             := addr(fserr);
      itemnum_array[2] := 0;
      itemstatus       := 0;

      AifProcPut( overall_status,
                  itemnum_array,
                  item,
                  itemstatus,
                  ,{my pin}
                  ,{my pid}
                  AIF_MY_VENDOR_ID );
      end
    else
      { Not a file equation:                                          }
      { do not skip the call to the target, that is, call fopen_nm() }
      stubout := FALSE;
end;

Begin
End.
```

Arm Handler (PASCAL)

```
{------------------------------------------------------------------------}
{ This program will arm the handler "user_handler" residing in "TESTXL.PE".}
{ The target is fopen_nm(), which is an internal procedure in the NL.      }
{                                                                          }
{ To compile -> :pasxl thisFile                                            }
{                                                                          }
{ To link    -> :link; cap=pm; xl='pexl.pub.sys'                           }
{                            NOTE: pexl must be the last library specified }
{                                  in the executable library list          }
{                                                                          }
{ To use with the handler provided as example, you must specify that the   }
{ handler is to be called BEFORE the target.                               }
{------------------------------------------------------------------------}

$standard_level 'Ext_Modcal'$

program Call_AIF_PEARM( input, output );
const
     AIF_MY_VENDOR_ID      = 1234;
     PE_INVOCATION_HANDLER = 0;
     PE_TERMINATION_HANDLER= 1;
     TAB                   = chr(9);
type
     status_type           = record case boolean of
                               true : ( all    : integer);
                               false: ( info   : shortint;
                                        subsys: shortint);
                             end;

     aifpe_file_name_type  = packed array [1..37] of char;
     aifpe_proc_name_type  = packed array [1..34] of char;
     aifpe_prod_name_type  = packed array [1..4]  of char;
var
     answer                : char;
     bequeathe             : boolean;
     bind_id               : integer;
     cm                    : boolean;
     handler_lib           : aifpe_file_name_type;
     handler_pri           : integer;
     handler_proc          : aifpe_proc_name_type;
     handler_type          : integer;
     overall_status        : status_type;
     pin                   : integer;
```

```
      prod_name              : aifpe_prod_name_type;
      target_lib             : aifpe_file_name_type;
      target_pri             : integer;
      target_proc            : aifpe_proc_name_type;


procedure GetPrivMode; intrinsic;
procedure GetUserMode; intrinsic;
procedure Terminate;   intrinsic;

$PUSH, SYSINTR 'peintr.pe.sys'$
procedure PEARM; intrinsic;
$POP$

Begin
  {Prompt to find PIN #, and if children processes are to be involved}

  prompt(
  'Enter the Pin# of the process for which the handler is to be armed:',TAB);
  readln(pin);

  prompt('Should the handler also be armed for the children of Pin #',
         pin:1,' (Y,N)[N]?',TAB);
  readln(answer);
  bequeathe := (answer in ['Y','y']);

  prompt('Is the handler to be called Before or After calling the target',
         '(B,A)[B]?',TAB);
  readln(answer);
  if (answer in ['A','a']) then handler_type := PE_TERMINATION_HANDLER
                           else handler_type := PE_INVOCATION_HANDLER;
  writeln;
```

```
{ initialization of PEARM() parameters }
overall_status.all := 0;
handler_proc       := '"user_handler"';  {name of handler procedure}
handler_lib        := '"TESTXL.PE"';     {library containing handler proc}
target_proc        := '"fopen_nm"';      {name of target procedure}
target_lib         := '"NL.PUB.SYS"';    {library containing target proc}
handler_pri        := -1;                {Will return the priority assigned.}
target_pri         := -1;                {It will be the next highest pri.  }
bind_id            := 0;                 {NL/XL.pub.sys have pri 1 and 2.   }
prod_name          := 'wxyz';            {s/w product or developer name}
cm                 := FALSE;             {target procedure CM?}

GetPrivMode;

PEARM( overall_status,
        handler_proc,
        target_proc,
        handler_lib,
        target_lib,
        handler_pri,
        target_pri,
        handler_type,
        bind_id,
        prod_name,
        pin,
        bequeathe,
        cm,
        AIF_MY_VENDOR_ID );

GetUserMode;

if (overall_status.all <> 0) then
   begin
   writeln('*** PEARM() returned an error status: ',
           '(info= ', overall_status.info:1,', ',
           'subsys= ',overall_status.subsys:1,')');
   Terminate;
   end;

writeln('Handler has been successfully armed.');
writeln('Priority #',target_pri:1,' has been assigned to ',target_lib);
writeln('Priority #',handler_pri:1,' has been assigned to ',handler_lib);

End.
```

Disarm Handler (PASCAL)

```
{-------------------------------------------------------------------------}
{ This program will disarm the handler "user_handler" residing in "TESTXL.PE".}
{ The target was fopen_nm(), which is an internal procedure in the NL.      }
{                                                                           }
{ To compile -> :pasxl thisFile                                             }
{                                                                           }
{ To link    -> :link; cap=pm; xl='pexl.pub.sys'                            }
{                            NOTE: pexl must be the last library specified  }
{                                  in the executable library list           }
{-------------------------------------------------------------------------}

$standard_level 'Ext_Modcal'$

program Call_AIF_PEDISARM( input, output );
const
     AIF_MY_VENDOR_ID      = 1234;
     PE_INVOCATION_HANDLER = 0;
     PE_TERMINATION_HANDLER= 1;
     TAB                   = chr(9);
type
     status_type           = record case boolean of
                               true : ( all    : integer);
                               false: ( info   : shortint;
                                        subsys: shortint);
                             end;

     aifpe_file_name_type  = packed array [1..37] of char;
     aifpe_proc_name_type  = packed array [1..34] of char;
     aifpe_prod_name_type  = packed array [1..4]  of char;
var
     answer                : char;
     bequeathe             : boolean;
     handler_lib           : aifpe_file_name_type;
     handler_proc          : aifpe_proc_name_type;
     handler_type          : integer;
     overall_status        : status_type;
     pin                   : integer;
     target_lib            : aifpe_file_name_type;
     target_proc           : aifpe_proc_name_type;
```

```
procedure GetPrivMode; intrinsic;
procedure GetUserMode; intrinsic;
procedure Terminate;   intrinsic;

$PUSH, SYSINTR 'peintr.pe.sys'$
procedure PEDISARM; intrinsic;
$POP$

Begin
  {Prompt to find PIN #, and if children processes are to be involved}
  prompt(
  'Enter the Pin# of the process for which the handler is to be disarmed:',TAB);
  readln(pin);

  prompt('Should the handler also be disarmed for children of Pin #',
         pin:1,' (Y,N)[N]?',TAB);
  readln(answer);
  bequeathe := (answer in ['Y','y']);

  prompt(
  'Was the handler called Before or After calling the target (B,A)[B]?',TAB);
  readln(answer);
  if (answer in ['A','a']) then handler_type := PE_TERMINATION_HANDLER
                           else handler_type := PE_INVOCATION_HANDLER;
  writeln;

  {initialization of PEDISARM() parameters }
  overall_status.all := 0;
  handler_proc       := '"user_handler"';
  handler_lib        := '"TESTXL.PE"';
  target_proc        := '"fopen_nm"';
  target_lib         := '"NL.PUB.SYS"';

  GetPrivMode;

  PEDISARM( overall_status,
            handler_proc,
            target_proc,
            handler_lib,
            target_lib,
            handler_type,
            pin,
            bequeathe,
            AIF_MY_VENDOR_ID );
  GetUserMode;
```

```
  if (overall_status.all <> 0) then
     begin
     writeln('*** PEDISARM() returned an error status: ',
             '(info= ', overall_status.info:1,', ',
             'subsys= ',overall_status.subsys:1,')');
     Terminate;
     end;

  writeln('Handler has been successfully disarmed.');
End.
```

Handler (C)

```
/*-------------------------------------------------------------------------*/
/* This program is an example of INVOCATION_HANDLER written in C/iX.       */
/* It intercepts HPFopen() to display the arguments that have been passed  */
/* by the caller.                                                          */
/* There are two "hidden" words that you should be aware of. One is the 1st */
/* argument passed to the handler. This is because of the Pascal/iX option */
/* "Option extensible ###".                                                */
/* The second one is the 1st argument passed to HPFopen(), since this proc */
/* can have a variable number of arguments. This "hidden" word is the number */
/* of arguments that have been actually passed (including the 2 required   */
/* parms).                                                                 */
/*                                                                         */
/* The handler defines the ptr 'hpfopen_args_ptr' and uses it to walk back */
/* the stack. Refer to the Pascal/iX handler for detailed explanations.    */
/*                                                                         */
/* Compile ->  :ccxl thisFile,,$Null;info='-Aa +02'                        */
/*                                                                         */
/* Link    ->  :linkedit                                                   */
/*             LinkEd> buildxl myxl; limit=1                               */
/*             Linked> addxl $oldpass; privlev=2;                          */
/*                           rl=libc.lib.sys, libcansi.lib.sys; merge      */
/*                     NOTE: rl=... needed because CI doesn't open XL.PUB.SYS*/
/*-------------------------------------------------------------------------*/
#include <stdio.h>

#define FALSE 0
#define TRUE  1

typedef unsigned char BOOLEAN;
typedef signed char   I8;
typedef short         I16;
typedef int           I32;

void hpfopen_handler( I32      hidden_word_for_opt_extensible,
                      I32      bindid,
                      BOOLEAN *stubout,
                      void    *parms_area )

{
I32 *hpfopen_args_ptr;
I32  nb_actual_args;

        hpfopen_args_ptr = parms_area;
        nb_actual_args   = *(--hpfopen_args_ptr);
```

```
        /* Check that 'nb_actual_args' is an even number within valid range. */
        /* Minimum is the 2 required parameters. Maximum number of args that */
        /* can be passed to hpfopen() is 84: 2 required + 2*41 optionals.    */

    if ((nb_actual_args % 2) || (nb_actual_args<2) || (nb_actual_args>84))
        {
           /* Something wrong here. let's stop here as far as the */
           /* handler is concerned, and call the target, hpfopen()*/
           stubout = FALSE;
           return;
        }

        /* Display the 2 required parameters (filenum, status) first. */
        printf("HPFOPEN(\n");
        printf("\t&filenum = %x,\n", *(--hpfopen_args_ptr));
        printf("\t&status  = %x,\n", *(--hpfopen_args_ptr));

        /* Display the (nb_actual_args - 2) optional arguments.     */
        /* Since each iteration displays two args (item_num, item), */
        /* we have to divide nb_actual_args by 2.                   */
        {
           I32 i;
           for (i=0, nb_actual_args -= 2; i < (nb_actual_args / 2); i++)
           {
                printf("\titem_num[%.2d] = %d,\n", i, *(--hpfopen_args_ptr));
                printf("\titem[%.2d]     = %x,  ", i, *(--hpfopen_args_ptr));

                /* Prints what is being pointed to by item in both      */
                /* hex and ascii (26 chars max) modes.                  */
                /* Check alignment before deferencing *hpfopen_args_ptr */
                if ((*hpfopen_args_ptr % 4) == 0)
                        /* 32-bits aligned */
                        printf("->  0x%.8x", *(I32 *)(*hpfopen_args_ptr));
                else
                        if ((*hpfopen_args_ptr % 2) == 0)
                        /* 16-bits aligned */
                        printf("->  0x%.8x", *(I16 *)(*hpfopen_args_ptr));
                        else
                        /* 8-bits aligned */
                        printf("->  0x%.8x", *(I8  *)(*hpfopen_args_ptr));
                        printf("  '%.26s'\n", (char *)(*hpfopen_args_ptr));
           }
        }
        printf("       )\n");
        /* This invocation handler is done now. Since stubout = FALSE  */
        /* (meaning don't skip the target), AIF:PE will call hpfopen().*/
     stubout = FALSE;
    }
```

Arm Handler (C)

```
/*-------------------------------------------------------------------*/
/* This C/iX program illustrates how to arm an AIF:PE handler.       */
/*                                                                   */
/* Compile -> :ccxl thisFile,,$Null;info='-Aa +02'                   */
/*                                                                   */
/* Link    -> :link; cap=pm; rl=^ccstdrl.lib.sys; xl='pexl.pub.sys' */
/*                NOTE: pexl must be the last library specified      */
/*                      in the executable library list               */
/*-------------------------------------------------------------------*/

#define _MPEXL_SOURCE
#include <stdio.h>
#include <string.h>
#include <errno.h>
#pragma intrinsic GETPRIVMODE mpe_GetPrivMode
#pragma intrinsic GETUSERMODE mpe_GetUserMode
#pragma intrinsic PROCINFO    mpe_ProcInfo
#pragma intrinsic_file "peintr.pe.sys"
#pragma intrinsic PEARM
#define FALSE 0
#define TRUE  1
#define AIF_MY_VENDOR_ID    1234
#define INVOCATION_HANDLER  0
#define TERMINATION_HANDLER 1
typedef unsigned char BOOLEAN;
typedef short         I16;
typedef int           I32;

BOOLEAN       bequeathe;
I32           bind_id;
BOOLEAN       cm;
I16           err1, err2;
char          handler_lib[37];
I32           handler_pri;
char          handler_proc[34];
I32           handler_type;
t_mpe_status  overall_status;
I16           parents_pin;
I32           pin;
char          prod_name[4];
char          target_lib[37];
I32           target_pri;
char          target_proc[34];
```

```
main()
{
        strcpy(handler_proc,"'hpfopen_handler'");
        strcpy(handler_lib ,"'MYXL'");
        strcpy(target_proc ,"'HPFOPEN'");
        strcpy(target_lib  ,"'NL.PUB.SYS");
        strcpy(prod_name   ,"test");

        handler_type = INVOCATION_HANDLER;
        handler_pri  = -1;
        target_pri   = -1;
        bind_id      =  0;
        bequeathe    = TRUE;
        cm           = FALSE;

        mpe_ProcInfo( &err1, &err2, 0, 2, &parents_pin );
        if (err1) {
                printf("*** ProcInfo() returned errors: %d,%d\n", err1, err2);
  exit(1);
        }
        pin = parents_pin;

 mpe_GetPrivMode();

  PEARM( &overall_status,
            handler_proc,
            target_proc,
            handler_lib,
            target_lib,
            &handler_pri,
            &target_pri,
            handler_type,
            bind_id,
            prod_name,
            pin,
            bequeathe,
            cm,
            AIF_MY_VENDOR_ID );

        mpe_GetUserMode();
        if (overall_status.word != 0) {
                printf("*** PEArm() returned status: (info=%d, subsys=%d)\n",
                       overall_status.decode.error_num,
                       overall_status.decode.subsys_num);
                exit(1);
        }
```

```
        printf("Handler has been successfully armed.\n");
        printf("Priority #%d assigned to %s\n",handler_pri,handler_lib);
        printf("Priority #%d assigned to %s\n",target_pri ,target_lib);
}
```

Disarm Handler (C)

```
/*-------------------------------------------------------------------*/
/* This C/iX program illustrates how to disarm an AIF:PE handler.    */
/*                                                                   */
/* Compile -> :ccxl thisFile,,$Null;info='-Aa +O2'                   */
/*                                                                   */
/* Link    -> :link; cap=pm; rl=^ccstdrl.lib.sys; xl='pexl.pub.sys' */
/*             NOTE: pexl must be the last library specified         */
/*                   in the executable library list                  */
/*-------------------------------------------------------------------*/

#define _MPEXL_SOURCE
#include <stdio.h>
#include <string.h>
#include <errno.h>
#pragma intrinsic GETPRIVMODE mpe_GetPrivMode
#pragma intrinsic GETUSERMODE mpe_GetUserMode
#pragma intrinsic PROCINFO    mpe_ProcInfo
#pragma intrinsic_file "peintr.pe.sys"
#pragma intrinsic PEDISARM

#define FALSE 0
#define TRUE  1

#define AIF_MY_VENDOR_ID    1234
#define INVOCATION_HANDLER  0
#define TERMINATION_HANDLER 1

typedef unsigned char BOOLEAN;
typedef short         I16;
typedef int           I32;

BOOLEAN       bequeathe;
I16           err1, err2;
char          handler_lib[37];
char          handler_proc[34];
I32           handler_type;
t_mpe_status  overall_status;
I16           parents_pin;
I32           pin;
char          target_lib[37];
char          target_proc[34];
```

```
main()
{
        strcpy(handler_proc,"'hpfopen_handler'");
        strcpy(handler_lib ,"'MYXL'");
        strcpy(target_proc ,"'HPFOPEN'");
        strcpy(target_lib  ,"'NL.PUB.SYS'");

        handler_type = INVOCATION_HANDLER;
        bequeathe    = TRUE;

        mpe_ProcInfo( &err1, &err2, 0, 2, &parents_pin );
        if (err1) {
                printf("*** ProcInfo() returned errors: %d,%d\n", err1, err2);
                exit(1);
        }
        pin = parents_pin;

        mpe_GetPrivMode();

        PEDISARM( &overall_status,
              handler_proc,
              target_proc,
              handler_lib,
              target_lib,
              handler_type,
              pin,
              bequeathe,
              AIF_MY_VENDOR_ID );

        mpe_GetUserMode();

        if (overall_status.word != 0) {
                printf("*** PEDISARM() returned status: (info=%d, subsys=%d)\n",
                       overall_status.decode.error_num,
                       overall_status.decode.subsys_num);
                exit(1);
        }

        printf("Handler has been successfully disarmed.\n");
}
```

A

AIF Status Messages

Architected Interface Facility: Procedure Exits status messages have a base value of -12000. Errors -12900 and beyond indicate internal operating system errors that should never occur, and SRs should be filed in the event they do.

PEARM and **PEDISARM** call the system intrinsics **HPGETPROCPLABEL** and **HPFOPEN** to validate procedures and library files. Errors from these intrinsics are passed directly to the user and hold the same meaning as found in the *MPE/iX Intrinsics Reference Manual* (32650-90028).

-12001	**MESSAGE**	AIF:PE not allowed on system.
	CAUSE	The :DISALLOW command has been issued on the system.
	ACTION	Use the :ALLOW command to allow AIF:PE.
-12002	**MESSAGE**	AIF:PE handler is not of correct privilege.
	CAUSE	Exec level is 3 or call level is less than target call level.
	ACTION	Set exec level <= 2 and call level >= target.
-12003	**MESSAGE**	Cannot unlock AIF:PE.
	CAUSE	No exclusive lock is in effect for AIF:PE.
	ACTION	Remove request to unlock AIF:PE.
-12004	**MESSAGE**	Incompatible target and handler priorities.
	CAUSE	Target library priority >= handler library priority.
	ACTION	Ensure target library priority < handler library priority.
-12005	**MESSAGE**	Handler from a temp file cannot be armed on other J/S domains.
	CAUSE	Attempt to arm handler from temp file on other J/S domain.
	ACTION	Save handler file as permanent file.

-12006	MESSAGE	Handler from a temp file cannot be armed on system processes.
	CAUSE	Attempt to arm handler from temp file on system process.
	ACTION	Save handler file as permanent file.
-12007	MESSAGE	A specified library cannot be a UDSL.
	CAUSE	Library already loaded as non-UDSL.
	ACTION	Arm handlers on other PINs before on self.
-12008	MESSAGE	Duplicate target/handler/pin binding.
	CAUSE	Attempt to arm existing target/handler/pin binding.
	ACTION	Remove duplicate binding request.
-12009	MESSAGE	Multiple handlers from same library not allowed on same target.
	CAUSE	Handlers for same target must have unique priorities/libraries.
	ACTION	Place handlers for same target in separate libraries.
-12010	MESSAGE	PIN specified during arming did not exist.
	CAUSE	State of PIN specified for arming was other than Process_Alive.
	ACTION	Ensure that PIN specified is active at time of arming.
-12011	MESSAGE	Target or handler library entry not found in AIF:PE tables.
	CAUSE	Binding no longer exists.
	ACTION	Check status of binding before attempting to disarm.
-12012	MESSAGE	Requested AIF:PE priority is not available.
	CAUSE	Requested AIF:PE priority is in use by another user.
	ACTION	Use alternative priority or AIF:PE default.
-12013	MESSAGE	Target or handler procedure entry not found in AIF:PE tables.
	CAUSE	Binding no longer exists.
	ACTION	Check status of binding before attempting to disarm.

-12014	MESSAGE	AIF:PE is not installed on the system.
	CAUSE	AIF:PE must be purchased and installed before execution.
	ACTION	Purchase and install AIF:PE product (P/N 36249A).
-12015	MESSAGE	Attempt to add entry to AIF:PE library table failed.
	CAUSE	AIF:PE library table is full; no more available entries.
	ACTION	Decrease use of AIF:PE on the system and enter a Service Request to increase the configured maximum number of AIF:PE libraries.
-12016	MESSAGE	Attempt to assign AIF:PE library priority failed.
	CAUSE	AIF:PE library table is full; no more available priorities.
	ACTION	Decrease use of AIF:PE on the system and enter a Service Request to increase the configured maximum number of AIF:PE libraries.
-12017	MESSAGE	Attempt to add entry to AIF:PE procedures table failed.
	CAUSE	AIF:PE procedures table is full; no more available entries.
	ACTION	Decrease use of AIF:PE on the system and enter a Service Request to increase the configured maximum number of AIF:PE procedures.
-12018	MESSAGE	Attempt to add AIF:PE binding entry failed.
	CAUSE	AIF:PE binding entry table full; no more available entries.
	ACTION	Decrease use of AIF:PE on the system and enter a Service Request to increase the configured maximum number of AIF:PE bindings.
-12019	MESSAGE	Attempt to add AIF:PE binding sequence header.
	CAUSE	AIF:PE binding header table full; no more available entries.
	ACTION	Decrease use of AIF:PE on the system and enter a Service Request to increase the configured maximum number of AIF:PE binding headers.
-12020	MESSAGE	Initialization of AIF:PE failed.
	CAUSE	Indicates resource shortage.
	ACTION	Check machine utilization; redistribute workload.
-12021	MESSAGE	Abort of AIF:PE failed.
	CAUSE	Indicates resource shortage.
	ACTION	Check machine utilization; redistribute workload.

-12022	MESSAGE	Initialization of AIF:PE process local data structures failed.
	CAUSE	Indicates resource shortage.
	ACTION	Check machine utilization; redistribute workload.
-12025	MESSAGE	Incompatible target mode (CM,NM) specified in arming.
	CAUSE	Target has been armed prior with opposite mode.
	ACTION	Specify opposite mode in arming.
-12026	MESSAGE	Invalid procedure library specified.
	CAUSE	Target or handler procedure was not found in associated library.
	ACTION	Ensure target/handler libraries contain corresponding routines.
-12027	MESSAGE	Cannot unquiesce the system.
	CAUSE	The system was not quiesced.
	ACTION	Remove request to unquiesce.
-12028	MESSAGE	Duplicate bequeathed binding specified.
	CAUSE	An attempt was made to bequeathe a duplicate binding to a PIN.
	ACTION	Remove request for duplicate bequeathe.
-12029	MESSAGE	Incompatible start image.
	CAUSE	AIF:PE millicode does not exist in start image.
	ACTION	Obtain correct start image from HP Support Representative.

B

AIF Data Structures

This appendix defines the type conventions and types used in this manual.

```
bit1                    =     0 .. 1;

bit2                    =     0 .. 3;

bit8                    =     0 .. 255;

bit14                   =     0 .. 16383;

bit16                   =     0 .. 65535;

bit31                   =     0 .. 2147483647;
```

```
aifpe_filename_type   =

     packed array [ 1 .. 37 ] of char ;          ( 0.0 @ 37.0 )

Array size: 37 bytes
Element size: 1 byte
Alignment : 1 byte

aifpe_filename_type contains a fully qualified file name of
filename.groupname.accountname. The first character
is assumed to be a delimiter for start and end of a character
array (for example,"NL.PUB.SYS").
```

```
aifpe_procname_type   =

     packed array [ 1 .. 34 ] of char ;          ( 0.0 @ 34.0 )

Array size: 34 bytes
Element size: 1 byte
Alignment : 1 byte

aifpe_procname_type contains a procedure name in the format
procname. The first character is assumed to be a delimiter
for start and end of a character array (for example,
"MYPROCNAME").
```

```
aifpe_prodname_type   =

     packed array [ 1 .. 4 ] of char ;           ( 0.0 @ 4.0 )

Array size: 4 bytes
Element size: 1 byte
Alignment : 1 byte
```

```
inaddr_type   =

    packed array [ 1 .. n ] of bit16 ;

Array size: 2n bytes
Element size: 2 bytes
Alignment : 4 bytes
```

```
modes_type   =

     packed array [ 1 .. n ] of integer ;

Array size: 4n bytes
Element size: 4 bytes
Alignment : 4 bytes
```

```
outaddr_type   =

     packed array [ 1 .. n ] of globalanyptr ;

Array size: 8n bytes
Element size: 8 bytes
Alignment: 8 bytes
```

```
status_type   =

     record
       case boolean of
         true  : ( all    : integer ) ;          ( 0.0 @ 4.0 )
         false : ( info   : shortint ;           ( 0.0 @ 2.0 )
                   subsys : shortint ) ;         ( 2.0 @ 2.0 )
     end ;

Record size : 4 bytes
Alignment   : 4 bytes
```

C

Handler Coding Requirements

This section describes the development of handler procedures. It outlines the calling sequence expected by Architected Interface Facility: Procedure Exits when it passes control to the handler. The handler may access the parameter list of the target, but to do so requires a knowledge of the calling sequence of the target procedure. This may require dissemination of proprietary information in certain instances, especially in the case of intercepting internal O/S procedures.

Calling Sequence of a Handler

The Architected Interface Facility: Procedure Exits product transfers control to user specified handler procedures upon call to the target procedure.

The following sections present the calling sequence of both invocation and termination handler procedures.

Invocation Handlers

Invocation handlers are called upon invocation of a target to which they are bound. Invocation handlers have the option of stubbing out the target procedure so that it is not executed. Instead, control is transferred to the first applicable termination handler. In the case where the target is a function, the handler is expected to initialize general register 28 (and general register 29) with the functional result.

An abnormal termination of an invocation handler will also cause the target to be stubbed out and the first applicable termination handler to be called. In the case of no termination handlers, the escape will be propagated to the user.

The layout of the parameters in the parms_area is exactly as it would be in a normal procedure call. Refer to the *Procedure Calling Conventions Reference Manual* (09740-90015) for details of parameter area layout. Language manuals may also be needed for the language in which the target procedure and/or the handler procedure(s) are coded. The Architected Interface Facility: Procedure Exits product allows handlers to be written in any HP supported language. For MPE iX procedures, the language of choice is Pascal.

Following is an example of the general calling sequence of a user invocation handler procedure USER_BEGINHANDLER:

```
procedure  USER_BEGINHANDLER (
                              bindid          :integer;
                          var stubout         :boolean;
                              parms_area      :localanyptr;
                          var func_rtn_area   :<func type, for example integer,
                                                                    shortint>;
                              )
Option extensible 4;
```

Parameter descriptions are as follows:

bindid (I32)	A user supplied id for the current target/handler binding.
stubout (B)	A value of True indicates that the target procedure should be stubbed out, or not executed. This will result in all remaining invocation handlers being skipped; all remaining invocation handlers will have priority less than the current handler. Control is then passed to the first termination handler with priority greater than or equal to this invocation handler. Note that invocation and termination handlers in XL.PUB.SYS and NL.PUB.SYS are always executed and are excepted from the above rule. A value of False means that this handler desires that the target procedure be executed, but it does not guarantee it. Remaining invocation handlers may elect to stub out the target call.
parms_area (@32)	A pointer to the parameters being passed to (from) the target procedure.
func_rtn_area (@32)	Valid only for functional targets. For NM targets, it can be interpreted as being the same as general register 28 at entry time. If the functional return is less than 64 bits, then func_rtn_area is a 32 bit pointer to the actual return value. If the functional return is greater than 64 bits, then func_rtn_area is a 32 bit pointer to the address of the actual return value. For CM targets, it can be interpreted as a pointer to the CM word before the stack marker (Q-4) of the target call.

Termination Handlers

Termination handlers are called upon termination of a target to which they are bound. In the event of an abnormal termination from an invocation handler or the target procedure itself, the first termination handler is notified and passed the escape code. It can then elect to propagate the escape code to the next handler, or stub out the escape, in which case the next handler will execute as if no error occurred.

If an escape code is propagated through all the termination handlers and is outstanding upon return from the last termination handler, then it will be propagated to the caller.

Following is an example of the general calling sequence of a user termination handler procedure USER_ENDHANDLER:

```
procedure  USER_ENDHANDLER (
                    bindid          : integer
                var escaped         : boolean;
                var escape_code     : integer;
                var func_rtn_area   : <func type, for example integer,
                                                          shortint>;
                    )
Option extensible 4;
```

Parameter descriptions are as follows:

bindid (I32)	A user supplied id for the current target/handler binding.
escaped (B)	A value of True indicates an unrecovered escape from either the target or an invocation handler and propagates the escape code to the next termination handler. A value of False indicates no escape occurred.
escape_code (I32)	Valid only if escaped is True. Holds the escape code associated with the escape.
func_rtn_area (@32)	Valid only for functional targets. For NM targets, it can be interpreted as being the same as general register 28 at exit time. If the functional return is less than 64 bits, then func_rtn_area is a 32 bit pointer to the actual return value. If the functional return is greater than 64 bits, then func_rtn_area is a 32 bit pointer to the address of the actual return value. For CM targets, it can be interpreted as a pointer to the last CM work of the functional return area. In order to successfully access and/or modify the return value, the user must know its full size and begin at the appropriate address.

Handler Declaration Examples

The following are examples of handler declarations in HP Pascal/iX.

Example One

In this example, the target procedure could be either an NM or CM procedure. The declarations are as follows:

```
procedure Begin_Handler (
                              bindid      :integer;
                          var stubout     :boolean;
                              parms       :localanyptr;
                             )
Option extensible 3;

procedure End_Handler (
                              bindid      :integer;
                          var escaped     :boolean;
                          var ecode       :integer;
                           )
Option extensible 3;
```

Example Two

In this example, the target procedure is an NM function that returns less than 64 bits. The declarations are as follows:

```
procedure Begin_Handler (
                          bindid      :integer;
                      var stubout     :boolean;
                          parms       :localanyptr;
                      var f_rtn       :<func type, for example integer,
                                                                shortint>;
                         )
Option extensible 4;

procedure End_Handler (
                          bindid      :integer;
                      var escaped     :boolean;
                      var ecode       :integer;
                      var f_rtn       :<func type, for example integer,
                                                                shortint>;
                       )
Option extensible 4;
```

Example Three In this example, the target procedure is an NM function that returns more than 64 bits. The declarations are as follows:

```
procedure Begin_Handler (
                            bindid          :integer;
                        var stubout         :boolean;
                            parms           :localanyptr;
                        var f_rtn_ptr       :<ptr to func type>;
                           )
Option extensible 4;

procedure End_Handler (
                            bindid          :integer;
                        var escaped         :boolean;
                        var ecode           :integer;
                        var f_rtn_ptr       :<ptr to func type>;
                        )
Option extensible 4;
```

Example Four In this example, the target procedure is an CM function. The declarations are as follows:

```
procedure Begin_Handler (
                        bindid          :integer;
                    var stubout         :boolean;
                        parms           :localanyptr;
                       )
Option extensible 3;

procedure End_Handler (
                        bindid          :integer;
                    var escaped         :boolean;
                    var ecode           :integer;
                    var f_rtn_ptr       :<ptr to last CM work of func type>;
                     )
Option extensible 4;
```

Note

For CM procedures or functions, the pointer parms will point to the last word of the CM parms area, which is Q-4. The pointer f_rtn_ptr will point to the last word of the CM functional return area.

For NM procedures or functions, the pointer parms will point to the next word after the NM parms area on the stack (DP).

D

Glossary

arming	The act of specifying a particular binding. A user arms a handler on a target to create a binding.
bequeathe	To pass down a process's bindings to any descendant processes (child processes, grandchild process, etc.). For example, a binding active for the system's root process (PIN=1) may be bequeathed in order to create a systemwide binding.
binding	A logical association retained by procedure exits between a handler procedure and a target procedure. A binding indicates what handler procedure(s) to execute upon invocation and/or termination of a target procedure.
callx	A system routine that facilitates an external procedure call. The Architected Interface Facility: Procedure Exits `callx` will actually transfer control to an Architected Interface Facility: Procedure Exits assembly procedure that will cause associated handler routines to be invoked.
CM instrumentation	For intercepting procedures from a Compatibility Mode program, this is the act of creating stubs for target procedures and modifying target procedures to call procedure exits AIFs.
Current priority	The priority of the most recently executed handler procedure for a given target. This is kept as the top of stack for an Architected Interface Facility: Procedure Exits maintained priority stack.
External call	A procedure call from a procedure in one SOM to a procedure in another SOM. May also be referred to as an Inter-SOM call.
Handler procedure	Procedure to be executed upon interception of a target procedure at invocation or termination.
Intercept	Execute, or transfer control to, a specified handler procedure upon invocation or termination of a target procedure.

Internal call	A procedure call from a procedure in a particular SOM to another procedure in the same SOM. May also be referred to as an Inter-SOM call.
Invocation handler	Handler procedure that is executed upon invocation of a target procedure.
NM instrumentation	For intercepting procedures from a Native Mode program, this is the act of linking a procedure in such a way that forces internal calls to the procedure into external calls.
Priority	An integer value that is assigned to a library containing handlers and/or targets. All handlers/targets in a particular library have the same priority. This value is used to order multiple handlers on the same target, as well as to prevent mutual recursion among handlers.
process local binding	A binding with process specific scope (i.e. a binding that is in effect for a specific process on a system). Defined for the caller of PEARM only.
process external binding	A binding that is bequeathed or defined for another process then the caller of PEARM.
Scope of binding	The set of processes for which a binding takes effect.
Stub out	To skip execution of a procedure. An invocation handler may stub out a target procedure. All remaining invocation handlers and termination handlers with priority less than or equal to the stubbing invocation handler will also be skipped.
System object module (SOM)	The smallest loadable unit of an executable library containing object code for a number of procedures.
System library	A library of executable routines that is loaded into every program file's environment. On MPE/iX, NL.PUB.SYS and XL.PUB.SYS are system libraries.
Target procedure	The procedure to be intercepted. Upon invocation or termination, control is passed to a specified handler procedure.
Termination handler	Handler procedure that is executed upon termination of a target procedure.

User defined system library (UDSL)	A library of handler and/or target procedures that is used in bindings with scope of more than one process. Architected Interface Facility: Procedure Exits ensures that any external references in such a library are resolvable only through `NL.PUB.SYS` so that it may be used for any process. Architected Interface Facility: Procedure Exits maintains an active reference to a `UDSL` until the library is no longer needed for target/handler bindings.
XRT entry	A system defined entry within the Loader Cross Reference Table allocated in a process' local space at load time for each external procedure called by a SOM. The XRT entry contains the address of a **`callx`** routine that will read the rest of the XRT entry and perform the external call. Architected Interface Facility: Procedure Exits modifies the **`callx`** address to point to an Architected Interface Facility: Procedure Exits **`callx`** routine.

Index