Resource Management Programmer's Guide

900 Series HP 3000 Computers

HEWLETT PACKARD

HP Part No. 32650-90024
Printed in U.S.A. 19871101

E1187

Printing History

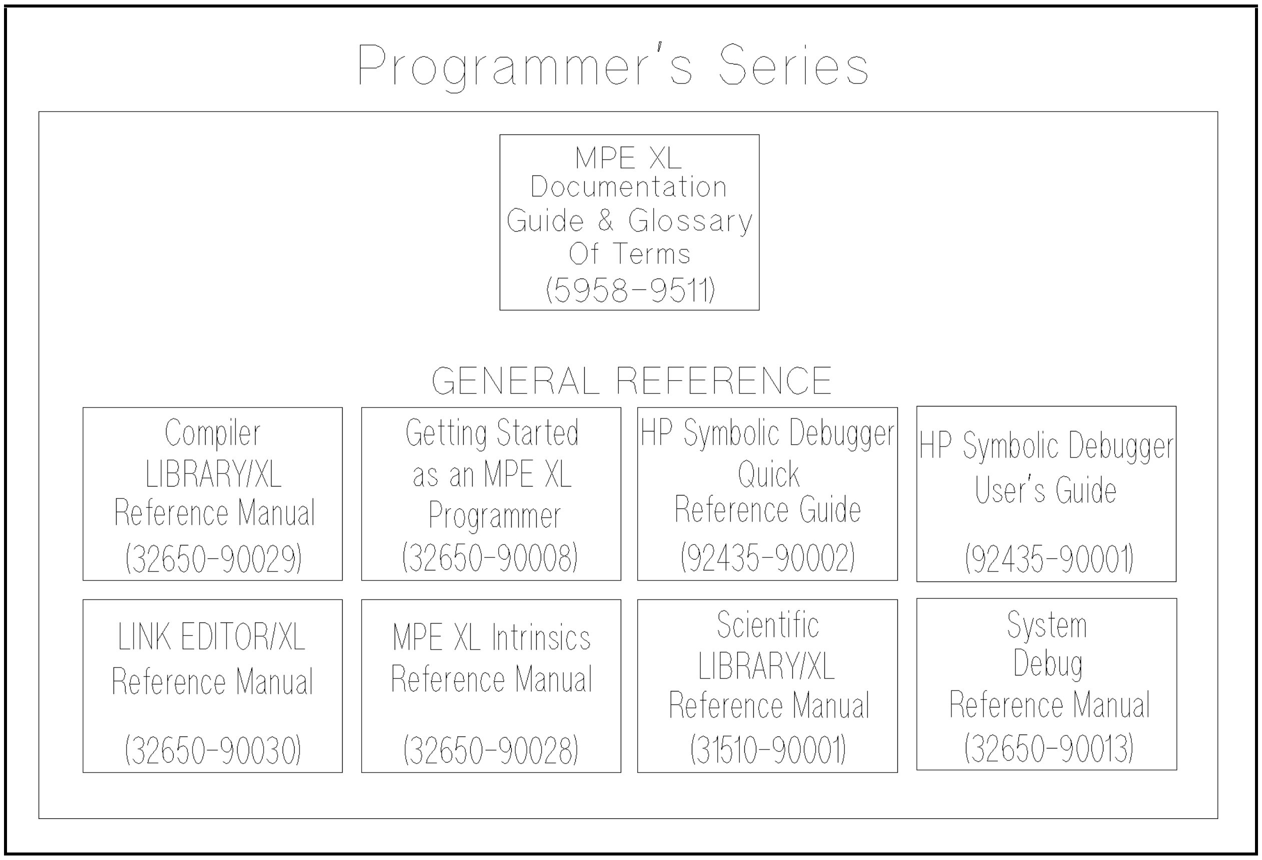
The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not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

Edition	Date	Software Version
First Edition	November 1987	A.01.00

List of Effective Pages The List of Effective Pages gives the date of the current edition, and lists the dates of all changed pages. Unchanged pages are listed as “ORIGINAL”. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition	November 1987

Documentation Map


Programmer's Series
MPE XL Documentation Guide & Glossary Of Terms (5958-9511)
GENERAL REFERENCE
Compiler LIBRARY/XL Reference Manual (32650-90029)
Getting Started as an MPE XL Programmer (32650-90008)
HP Symbolic Debugger Quick Reference Guide (92435-90002)
HP Symbolic Debugger User's Guide (92435-90001)
LINK EDITOR/XL Reference Manual (32650-90030)
MPE XL Intrinsics Reference Manual (32650-90028)
Scientific LIBRARY/XL Reference Manual (31510-90001)
System Debug Reference Manual (32650-90013)

Programmer's Series (con't)

PROGRAMMING TECHNIQUES

Accessing Files Programmer's Guide (32650-90017)	Command Interpreter Access & Variables Programmer's Guide (32650-90011)	Data Types Conversion Programmer's Guide (32650-90015)	Getting System Information Programmer's Guide (32650-90018)
Interprocess Communication Programmer's Guide (32650-90019)	Message Catalog Programmer's Guide (32650-90021)	Native Language Programmer's Guide (32650-90022)	Process Management Programmer's Guide (32650-90023)
Resource Management Programmer's Guide (32650-90024)	SORT-MERGE/XL Programmer's Guide (32650-90080)	Trap Handling Programmer's Guide (32650-90026)	User Logging Programmer's Guide (32650-90027)

Preface

Resource Management Programmer's Guide (32650-90024) is written for an experienced programmer who has a working knowledge of MPE/iX and is familiar with:

- A text editor
- At least one programming language
- Compiling, linking, and executing a program on MPE/iX

This manual contains detailed instructions describing how you can use system intrinsics within your application to accomplish two resource management tasks available through MPE/iX:

- Managing Shared Resources with Resource Identification Numbers (RINs)
- Dynamically loading procedures located in executable libraries (XLs)

This manual is part of the MPE/iX Programmer's Series. This series consists of the *MPE/iX Intrinsics Reference Manual* (32650-90028) and a set of task-oriented programmer's guides. Refer to the MPE/iX Programmer's Series Documentation Map for a description of how this manual relates to the rest of the series.

This manual contains the following chapters:

Chapter 1	**Introduction** provides you with an overview of the contents of this manual.
Chapter 2	**Managing Shared Resources with RINs** describes how you can use Resource Identification Numbers (RINs) to manage a specific resource shared by a set of jobs or sessions, so that no two jobs or sessions can use the resource at the same time.
Chapter 3	**Dynamic Loading of Library Procedures** describes how you can dynamically bind and load a procedure located in an executable library (XL).
Appendix A	**Global RIN Program Example** is an HP Pascal/XL program illustrating how you can use RIN management intrinsics to guarantee exclusive access to a selected record in a shared file.
Appendix B	**Dynamic Loading Program Example** is an HP Pascal/XL program illustrating how you use the `HPGETPROCPLABEL` intrinsic to dynamically load a procedure located in an executable library.

Conventions

NOTATION	DESCRIPTION
`UPPERCASE`	Within syntax statements, characters in uppercase must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example: `SHOWJOB` Valid entries: `showjob` `ShowJob` `SHOWJOB` Invalid entries: `shojwob` `ShoJob` `SHOW_JOB`
italics	Within syntax statements, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release: `RELEASE` *filename*
punctuation	Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.
`{ }`	Within syntax statements, braces enclose required elements. When several elements within braces are stacked, you must select one. In the following example, you must select `ON` or `OFF`: $\texttt{SETMSG}\begin{Bmatrix}\texttt{ON}\\\texttt{OFF}\end{Bmatrix}$
`[ ]`	Within syntax statements, brackets enclose optional elements. In the following example, brackets around `,TEMP` indicate that the parameter and its delimiter are optional: `PURGE {`*filename*`} [,TEMP]` When several elements with brackets are stacked, you can select any one of the elements or none. In the following example, you can select *devicename* or *deviceclass* or neither: $\texttt{SHOWDEV}\begin{bmatrix}\textit{devicename}\\\textit{deviceclass}\end{bmatrix}$

NOTATION — **DESCRIPTION**

`[ ... ]`

Within syntax statements, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:

[,*itemname*][...]

If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:

[*itemname*][, ...]

`| ... |`

Within syntax statements, a horizontal ellipsis enclosed in parallel vertical lines indicates that you can select more than one element that appears within the immediately preceding pair of brackets or braces. However, each element can be selected only one time. In the following example, you must select `,A` or `,B` or `,A,B` or `,B,A` :

$$\left\{\begin{array}{c}\texttt{,A}\\ \texttt{,B}\end{array}\right\}\texttt{| ... |}$$

If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, you must select `A` or `B` or `AB` or `BA`. The first element cannot be preceded by a comma:

$$\left\{\begin{array}{c}\texttt{A}\\ \texttt{B}\end{array}\right\}\texttt{|, ... |}$$

`...`

Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.

`␣`

Within syntax statements, the space symbol ␣ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank:

`SET[(`*modifier*`)]␣(`*variable*`);`

`shading`

Within an example of interactive dialog, `shaded` characters indicate user input or responses to prompts. In the following example, `OMEGA` is the user's response to the `NEW NAME` prompt:

`NEW NAME? OMEGA`

NOTATION	DESCRIPTION
□	The symbol □ indicates a key on the terminal's keyboard. For example, CTRL indicates the Control key.
CTRL *char*	CTRL *char* indicates a control character. For example, CTRL `Y` means you have to simultaneously press the Control key and the Y key on the keyboard.
base prefixes	The prefixes `%`, `#`, and `$` specify the numerical base of the value that follows: `%`*num* specifies an octal number. `#`*num* specifies a decimal number. `$`*num* specifies a hexadecimal number. When no base is specified, decimal is assumed.

Bit (*bit:length*) When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format Bit (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, `Bits (13:3)` indicates bits 13, 14, and 15:

```
most significant                least significant

|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0|  |  |  |  |  |  |  |  |  |  |  |  |13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Bit (0:1)                              Bits(13:3)
```

Contents

Figures

1

Introduction

Any element of an HP 3000 computer system that your program can access through MPE XL can be considered a resource. A resource can be an input or output device, a file, a subroutine, a library procedure, or a private data area.

A major function of MPE XL is to manage resources in such a way as to make your programming efforts easier and more efficient. MPE XL also provides you with system intrinsics to accomplish various tasks related to managing the resources your program may access.

This manual describes two such resource management tasks available to you through the operating system:

- Managing shared resources with Resource Identification Numbers (RINs).
- Dynamic loading of executable library procedures.

All intrinsics discussed in this manual is described in greater detail in the *MPE XL Intrinsics Reference Manual* (32650-90028). Commands discussed in this manual are described in the *MPE XL Commands Reference Manual* (32650-90003).

Managing Shared Resources with RINs

Chapter 2 describes how you can use RINs to manage a specific resource shared by a set of jobs or processes, so that no two jobs or processes can use the resource at the same time. This chapter discusses the following system commands and intrinsics:

- :`GETRIN` command
- :`FREERIN` command
- `LOCKGLORIN` intrinsic
- `UNLOCKGLORIN` intrinsic
- `GETLOCRIN` intrinsic
- `LOCKLOCRIN` intrinsic
- `UNLOCKLOCRIN` intrinsic
- `LOCRINOWNER` intrinsic
- `FREELOCRIN` intrinsic

Appendix A contains a program illustrating the use of the `LOCKGLORIN` and `UNLOCKGLORIN` intrinsics to guarantee exclusive access to selected records in a data file.

Dynamic Loading of Library Procedures

Chapter 3 describes how you can dynamically bind and load a procedure located in an executable library (XL). This chapter discusses the following system intrinsics:

- `HPGETPROCPLABEL` intrinsic
- `HPFIRSTLIBRARY` intrinsic
- `HPMYFILE` intrinsic
- `HPMYPROGRAM` intrinsic

Appendix B contains a program example illustrating the use of the `HPGETPROCPLABEL` intrinsic.

2

Managing Shared Resources with RINs

When you are developing an application you may wish to manage a specific resource that is being shared by a set of processes in a way that guarantees that one process at a time has exclusive access to that resource. MPE XL provides Resource Identification Numbers (RINs) that enable you to manage shared resources.

You can use RINs described in this chapter to manage anything you may consider a resource to your program, be it a device, a portion of a file, or a section of code in your program. In addition, the operating system provides a resource management scheme similar to RIN management through the use of the FLOCK and FUNLOCK intrinsics to guarantee your process exclusive access to a file being shared by a set of processes located in different jobs/sessions. Refer to *Accessing Files Programmer's Guide* (32650-90017) for details on using FLOCK and FUNLOCK.

A RIN is not assigned by MPE XL to any particular resource. The association of a RIN and a resource is established by cooperating programmers whose programs are sharing the resource. The RIN value is known to the operating system, but the resource with which it is associated is known only to you and other programmers who have agreed to manage the resource through RIN management.

Within the programs being executed by different processes, you and other programmers must first agree to associate a RIN to a particular resource. When the process executing your program seeks exclusive access to that resource, the process must successfully lock the associated RIN prior to accessing the resource. Successfully locking the RIN means that your process can access the resource exclusively, so long as the RIN remains locked.

If the attempt to lock the RIN is unsuccessful, it means that the RIN is locked and another process has exclusive access to the associated resource.

Note The successful management of resources using RINs is predicated upon the assumption that all accessors of a particular resource have agreed to access the resource through RIN-locking intrinsics. If a process does not use this RIN management scheme to access the resource, exclusive access to that resource cannot be guaranteed.

Multiple RIN (MR) Capability

There are two types of RIN available for your use:

- Global RIN, used to manage a resource shared by processes located in different jobs/sessions.
- Local RIN, used to manage a resource shared by processes located in the same job/session.

If you have standard user capabilities you can lock more than one local RIN at the same time. However, you can lock only one global RIN at a time, because of the danger of a system deadlock resulting from the improper use of global RINs. Refer to the discussion of deadlock found later in this chapter.

If your program needs to have two or more global RINs locked at the same time, you must have the Multiple RIN (MR) capability assigned by a System Manager or Account Manager to the group in which your program file resides. In addition, you must assign to your program file the MR capability-class attribute at link time using the `;CAP=` parameter of the `:LINK` command.

Because the operating system uses a resource management scheme similar to global RINs in the `FLOCK` and `FUNLOCK` intrinsics, you must count each active `FLOCK` call in your program as a locked global RIN.

Managing with Global RINs

A RIN used to manage a resource being shared by unrelated processes is called a global RIN. You use global RINs when you are using RIN management to prevent simultaneous access to a resource by two or more processes that may not be located in the same job/session. Each global RIN is a positive integer that is unique within MPE XL. Global RINs are acquired and released through system commands, and locked and unlocked through system intrinsics.

If your program has only standard user capabilities, it can lock only one global RIN (used at the unrelated process level) at a time. With MR capability, your program can lock two or more global RINs at the same time.

Acquiring Global RINs

You can acquire a global RIN with the `:GETRIN` command. Following is an example of a `:GETRIN` call:

```
:GETRIN PASSWORD
```

where `PASSWORD` is a required password, a character string of up to eight alphanumeric characters, beginning with an alphabetic character. You use the RIN password to restrict the locking of global RINs to authorized users.

Before you and other users can engage in global RIN management you must distribute the global RIN and its password to the other users. The user that acquires the global RIN is considered the owner of that RIN.

You typically enter the `:GETRIN` command during a session when you decide to use global RINs in your program to manage a resource. As a result of the command, MPE XL makes a global RIN available for use from a pool of free global RINs, and displays the global RIN in the following manner:

`RIN:` *rinnum*

Cooperating processes can use the RIN during the current session or during future jobs/sessions. A global RIN is available even when the owner is logged off. The global RIN is available until the owner of the RIN releases the RIN back to MPE XL with the `:FREERIN` command.

The total number of global RINs that MPE XL can assign is specified when the system is configured, but can never exceed 1024. If all currently available global RINs are acquired by users, the operating system rejects your request and issues the following message:

```
RIN TABLE FULL
```

In this case you must wait until a global RIN becomes available, or request that your System Manager raise the maximum number of global RINs that MPE XL can assign.

For additional information about the `:GETRIN` command, refer to the *MPE XL Commands Reference Manual* (32650-90003).

Locking and Unlocking Global RINs

Your process can lock a global RIN using the `LOCKGLORIN` intrinsic. Once you have successfully locked the RIN, no other process can lock the same global RIN until you either unlock it with the `UNLOCKGLORIN` intrinsic, or your process terminates.

While you have the global RIN locked, you are guaranteed exclusive access to the resource associated with the global RIN, so long as other processes first attempt to lock the same global RIN, prior to accessing the resource.

Following is an example of a `LOCKGLORIN` intrinsic call:

```
      .
      .
      .
RINNUM := 4;
LOCKCOND := 1;
RINPASSWORD := 'RIN4LOCK  ';
```

```
        LOCKGLORIN (RINNUM,LOCKCOND,RINPASSWORD);
              .
              .
              .
```

The parameters specified in the example are described below.

`RINNUM`	Passes the global RIN number associated with the resource you wish exclusive access to. The value 4 is a global RIN number returned by the `:GETRIN` command.
`LOCKCOND`	`LOCKCOND` passes a value indicating the following: if the global RIN is currently locked by another process, control does not return from the `LOCKGLORIN` call to your program until the global RIN is again available and successfully locked by your intrinsic call.
`RINPASSWORD`	`RINPASSWORD` passes the RIN password assigned to the global RIN through the `:GETRIN` command.

You use the `UNLOCKGLORIN` intrinsic to unlock a global RIN your process has previously locked with `LOCKGLORIN`. Once your process unlocks the RIN, it is available to be locked by other `LOCKGLORIN` calls.

Following is an example of an `UNLOCKGLORIN` intrinsic call:

```
                 .
                 .
                 .
            RINNUM := 4;
           UNLOCKGLORIN (RINNUM)
                 .
                 .
                 .
```

The parameter specified in the above example is described below.

`RINNUM`	Passes the global RIN associated with the resource you no longer wish exclusive access to. To unlock this global RIN, you must have previously locked it using `LOCKGLORIN`. The value 4 is a global RIN returned by the `:GETRIN` command.

Appendix A contains a program that uses global RINs to manage access to records in a data file being shared among multiple readers/writers. The program can be considered to be part of a book record maintenance application used in a library. Specifically, the program updates the location field of a book record located in the data file.

Consider, also, that the library has several sites where users can check out books. A librarian at each site has a terminal logged on to a session running the maintenance application. Different sessions, then, must be able to share access to the same data file for the purposes of updating book records.

Anytime a book is checked in or checked out from any site, the data file is updated to reflect the new location of the book. For example, if a book is checked in, the librarian who receives the book must update the record associated with the book, changing the location from `LOANED CARD# 451, DUE APRIL 1` to `AVAILABLE` so that the records reflect the current location of the book.

A problem exists maintaining the integrity of the records when two or more librarians access the same record simultaneously. For example, if two librarians, "A" and "B", access a book record simultaneously, the following may occur if provisions are not made to guarantee exclusive access to a record during updates:

1. "A" copies the book record into her stack, showing the book is available.
2. "B" simultaneously copies the same record into his stack, showing the book is available. This can occur because the data file is opened with the access type option of `HPFOPEN/FOPEN` set to SHARE (any other process, in any other session, can concurrently access this file).
3. "A" updates the location field of the record showing the book to be checked out, then posts the record to disc.
4. "B" updates the location field of the record showing a 24-hour hold on the book, then posts the record to disc.

The final result of this sequence is "B" overwriting the updated location entered by "A". The true location of the book has been lost. What should have occurred is that "B" should not have been able to access the record for the purposes of updating until "A" was finished with the update process.

Appendix A shows a program using the LOCKGLORIN and UNLOCKGLORIN intrinsics to ensure exclusive access to book records during an update. The program allows a user to lock four records in a file so that a record can be updated without chance of another user updating it simultaneously. In the program, the other users are not suspended when attempting to access records elsewhere in the file.

The file BOOKFILE, illustrated in Example 2-1, contains the titles and status of the 20 books in a library. The program in Appendix A uses this file as its data file.

```
TITLE: THE BORROWERS              FACULTY LOAN - DR. JOHNSON
TITLE: ALICE IN WONDERLAND        FACULTY LOAN - DR. JOHNSON
TITLE: PETER PAN                  AVAILABLE
TITLE: JUNGLE BOOK                AVAILABLE
TITLE: THE LIFE OF MERENB         AVAILABLE
TITLE: INTRO TO TAI CHI CHUAN     AVAILABLE
TITLE: TOM SAWYER                 LOANED CARD# 275, DUE APRIL 16
TITLE: TREASURE ISLAND            INTERLIBRARY LOAN - COUNTY LIBRARY
TITLE: A CHRISTMAS CAROL          AVAILABLE
TITLE: THE WIZARD OF OZ           AVAILABLE
TITLE: THE DARK CRYSTAL           AVAILABLE
TITLE: SPEED RACER                LOANED CARD# 921, DUE MARCH 25
TITLE: ULTRAMAN GOES TO TOWN      AVAILABLE
TITLE: H.M.S. PINAFORE            AVAILABLE
TITLE: FEAR OF FLYING             FACULTY LOAN - DR. STRANGELOVE
TITLE: SNOW WHITE                 FACULTY LOAN - DR. CHARMING
TITLE: DR. DOOLITTLE              INTERLIBRARY LOAN - ACME UNIV.
TITLE: TALES OF MOTHER GOOSE      AVAILABLE
TITLE: AESOP'S FABLES             AVAILABLE
TITLE: THE GULAG ARCHIPELAGO      LOANED CARD #36, DUE MAY 11
```

Figure 2-1.

Example 2-1. BOOKFILE Contents.

BOOKFILE contains 20 records, so the program must acquire five global RINs. (The program uses four records per global RIN). This is accomplished by repeatedly issuing the command:

```
:GETRIN BOOKRIN
```

BOOKRIN is the rinpassword specified in the program to lock the global RIN. Because MPE XL does not always assign global RINs in sequence, and because the program requires consecutive RINs to keep track of them more easily, it may be necessary to enter more :GETRIN commands before the program is first run in order to acquire the five consecutive global RINs. Extra RINs can be released with the :FREERIN command. For the purposes of this example, we assume that RINs 1 through 5 have been assigned.

The program in Appendix A uses the following procedures to accomplish its task:

- Procedure `error_ handler` is a standard error handling routine that is invoked whenever an intrinsic call is unsuccessful.
- Procedure `initialize_variables` initializes appropriate global variables prior to use.
- Procedure `open_files` opens the three files required by the program: `$STDIN`, `$STDLIST`, and `BOOKFILE`.
- Procedure `update_book_information` is the main procedure called after files have been opened. This procedure prints the program header to `$STDLIST`, then repeatedly prompts the user for a record to update (procedure `select_record`), until the user presses Return, instead of a record number, to indicate the end of the program. Each time a record is selected, the associated RIN is computed, then procedure `access_record_exclusively` is called.
- Procedure `access_record_exclusively` (Example 2-2) locks the global RIN associated with the selected record before calling procedure `update_record` to update the selected record. When procedure `update_record` is finished, the global RIN is unlocked before the program prompts the users for the next record number.
- Procedure `update_record` prints the selected record to `$STDLIST`, prompts the user for a new location, then reads the input from `$STDIN`. If the user supplies a new location, the record is updated, then immediately posted to disc.

```
procedure access_record_exclusively(rinnum:shortint);
  begin
  lockglorin(rinnum,lockflag,rinpassword);  {Lock global RIN          }
  if ccode <> CCE then error_handler(-1,103);
  freaddir(booklist,bookrecord,-72,accno);  {Read selected bookrecord }
  if ccode = CCL then error_handler(booklist,104) else
  if ccode = CCE then update_record;    {Call PROCEDURE update_record }
  unlockglorin(rinnum);                 {Unlock global RIN            }
  if ccode <> CCE then error_handler(-1, 110);
  end;
```

Figure 2-2.

Example 2-2. Procedure Access_Record_Exclusively.

Once the user selects a valid book record, and the correct RIN is computed, the program calls the intrinsic `LOCKGLORIN` to lock the RIN. If the RIN is already locked (by another process executing the same code to update a record), the record cannot be accessed until the RIN is unlocked by the process that first locked it.

Example 2-3 is a sample of an interactive session with the record update program located in Appendix A. Updated entries are accessed a second time to confirm successful modification.

```
:RUN BKUPDATE

LIBRARY INFORMATION PROGRAM.
ACCESSION NO:  4
TITLE:  THE LIFE OF MERENB        INTERLIBRARY LOAN - UNIV. OF OZ
NEW LOCATION  AVAILABLE
ACCESSION NO:  4
TITLE: THE LIFE OF MERENB         AVAILABLE
NEW LOCATION  (Return)
ACCESSION NO:  1
TITLE:  ALICE IN WONDERLAND       LOANED CARD# 451, DUE APRIL 1
NEW LOCATION  FACULTYLOAN-DR.JOHNSON
ACCESSION NO:  1
TITLE: ALICE IN WONDERLAND        FACULTY LOAN - DR. JOHNSON
NEW LOCATION  (Return)
ACCESSION NO:  18
TITLE: AESOP'S FABLES             AVAILABLE
NEW LOCATION  (Return)
ACCESSION NO:  (Return)

END OF PROGRAM
:
```

Figure 2-3.

Example 2-3. Execution of Record Update Program.

Releasing Global RINs

If you are the owner of a global RIN, you can use the `:FREERIN` command to release the global RIN back to the pool of free global RINs maintained by the operating system.

Following is an example of a `:FREERIN call`:

```
:FREERIN 8
```

where `8` is the global RIN you want released.

Note You should be certain that all other users of a global RIN are finished using the RIN before you release it back to MPE XL.

For additional information about the `:FREERIN` command, refer to the *MPE XL Commands Reference Manual* (32650- 90003).

Managing with Local RINs

A RIN used to manage a resource being shared by related processes is called a local RIN. You use local RINs to prevent simultaneous access to a resource by two or more processes in the same job/session. Each local RIN is a positive integer that is unique within your job/session.

Local RINs are assigned with the `GETLOCRIN` intrinsic, managed with the `LOCKLOCRIN` and `UNLOCKLOCRIN` intrinsics, and released with the `FREELOCRIN` intrinsic.

Acquiring Local RINs

You must acquire local RINs with the `GETLOCRIN` intrinsic before you can use them within your job/session. The following intrinsic call,

```
GETLOCRIN (6);
```

acquires six local RINs, RIN numbers 1 through 6, that can be used by the calling process as well as other processes in the same job/session.

All local RINs you are planning to use in your program must be acquired in just one call to `GETLOCRIN`. If your program requires additional local RINs after the initial `GETLOCRIN` call, you must first release all local RINs then acquire the new number of local RINs with another `GETLOCRIN` call.

Locking and Unlocking Local RINs

You can lock a local RIN using the `LOCKLOCRIN` intrinsic. Once your process has successfully locked the RIN, no other process can lock the same local RIN until your process unlocks it with the `UNLOCKLOCRIN` intrinsic.

While you have the local RIN locked, exclusive access is guaranteed only if other processes first attempt to lock the local RIN prior to accessing the resource associated with the locked RIN.

A local RIN, acquired by your program, can be locked and unlocked by any process in your program's process structure.

Following is an example of a `LOCKLOCRIN` intrinsic call from a program that has previously acquired local RINs 1 through 4:

```
          .
          .
          .
RINNUM := 4;
LOCK := 1;
LOCKLOCRIN (RINNUM,LOCK);
          .
          .
          .
```

The parameters specified in the example are described below.

`RINNUM`	Passes the local RIN number associated with the resource you wish exclusive access to. The value 4 is a local RIN returned by the `GETLOCRIN` intrinsic.
`LOCKCOND`	`LOCKCOND` passes a value indicating the following: if the local RIN is currently locked by another process, control does not return to your program from the `LOCKLOCRIN` call until the local RIN is again available and successfully locked by your process.

You use the `UNLOCKLOCRIN` intrinsic to unlock a local RIN that has been previously locked by the calling process.

Example 2-4 illustrates how the `LOCKLOCRIN` and `UNLOCKLOCRIN` intrinsics can be used by two processes executing in the same job/session. Assume that both the parent process (`PARENT`) and the child process (`CHILD`) are executing concurrently, line by line.

```
     PARENT PROCESS                     CHILD PROCESS

{  PARENT BEGINS EXECUTION    }
          .
          .
          .
     HPFOPEN (LP,STATUS,...);
     GETLOCRIN (3);
     LPRIN := 1;
     FWRITE (LP,...);
     CREATE (PROGNAME,,CHILD...);
     LOCKLOCRIN (LPRIN,1);
     ACTIVATE (CHILD);                  {   CHILD BEGINS EXECUTION    }
     FWRITE (LP,...);                             .
          .                                       .
          .                                  LPRIN := 1;
          .                                  HPFOPEN (LP,STATUS,...);
          .                                  LOCKLOCRIN (LPRIN,1);
          .
          .                              CHILD IS BLOCKED WHEN IT
          .                              ATTEMPTS TO LOCK RIN.
          .
          .                              EXECUTION CONTINUES WHEN RIN
          .                              IS UNLOCKED BY PARENT AND
          .                              LOCKED BY CHILD.
     UNLOCKLOCRIN (LPRIN);
          .                                       .
     LOCKLOCRIN (LPRIN,1);                        .
                                                  .
 PARENT IS BLOCKED WHEN IT                   FWRITE (LP,...);
 ATTEMPTS TO LOCK RIN.                            .
                                                  .
 EXECUTION CONTINUES WHEN RIN                     .
 IS UNLOCKED BY CHILD AND                         .
 LOCKED BY PARENT.                                .
                                             UNLOCKLOCRIN (LPRIN);
     FWRITE (LP,...);                             .
          .                                       .
          .                                       .
          .                                       .
```

Figure 2-4.

Example 2-4. Locking and Unlocking Local RINs.

In Example 2-4, both processes have agreed to RIN management, associating RIN 1 (designated in the program as `LPRIN`) with a line printer (designated as `LP`). When `PARENT` first accesses `LP`, `CHILD` has not been created, and so RIN management is not yet required.

To guarantee exclusive access to `LP` first, `PARENT` locks `LPRIN` before `CHILD` is activated. When `PARENT` finishes with `LP`, it unlocks `LPRIN`, thus making `LPRIN` available to be locked. In this case, `CHILD` has been blocked and is waiting for `LPRIN` to become available.

When `CHILD` attempts to lock `LPRIN`, `CHILD` passes a value to `LOCKLOCRIN` indicating that execution should be blocked until `LOCKLOCRIN` can successfully return the locked RIN. The blocking occurs because `PARENT` currently has `LPRIN` locked. Execution continues only when `PARENT` unlocks `LPRIN`, thus making RIN 1 available to be locked by `CHILD`. While `CHILD` has `LPRIN` locked, `PARENT` is unable to lock `LPRIN` until it is unlocked by `CHILD`.

Releasing Local RINs

You can use the `FREELOCRIN` intrinsic to release all local RINs your program previously acquired with `GETLOCRIN`. Following is an example of a call to `FREELOCRIN`:

```
FREELOCRIN;
```

Any process in your program's process structure can release local RINs. If you do not use `FREELOCRIN` to release local RINs, they are released to MPE XL when your program terminates.

Identifying a Local RIN Locker

The `LOCRINOWNER` intrinsic identifies the process in your program's process structure that has a particular local RIN locked. If the RIN is locked by the parent of the calling process, `LOCRINOWNER` returns a zero. If the RIN is locked by any other process, `LOCRINOWNER` returns the Process Identification Number (PIN) of that process.

Knowing the identity of the locking process is useful when parent and child processes are synchronizing access to one another through calls to the `ACTIVATE` and `SUSPEND` intrinsics.

Example 2-5 is an example of RIN management where a parent process (`PARENT`) acts as a monitor for several child processes (one of whom is identified as `CHILD1`). Assume that both `PARENT` and `CHILD1` are executing concurrently, line by line.

Note that two agreements have been made by the programmer regarding RIN management prior to writing the code in Example 2-5.

- When a child process wishes to communicate with `PARENT` it must first successfully lock local RIN 1 (designated in the program as `WHICHCHILD`). This guarantees that other child processes cannot interfere in the communication being performed while local RIN 2 (designated as `SYNCHRIN`) is locked.
- A child process must successfully lock local RIN 2 only after it has successfully locked local RIN 1. `PARENT` locks RIN 2 to guarantee that the child process that activated `PARENT` is suspended while `PARENT` executes code in the WHILE loop.

```
  PARENT PROCESS                     CHILD PROCESS

{ PARENT EXECUTING     }               {    CHILD EXECUTING     }
         .                                         .
    GETLOCRIN (2);                                 .
         .                              LOCKLOCRIN (WHICHCHILD,1);
    CHILDCOUNT := 0;                    LOCKLOCRIN (SYNCHRIN,1);
    WHILE CHILDCOUNT <= MAXCOUNT DO                .
    BEGIN                                          .
    SUSPEND (CHILDWAIT,SYNCHRIN);                  .
                                                   .
      PARENT IS SUSPENDED                          .
       UNTIL ACTIVATED                             .
        BY A CHILD                                 .
                                        PARENT := FATHER;
                                        ACTIVATE (PARENT);
       .                                SUSPEND (PARENTWAIT,SYNCHRIN);
    LOCKLOCRIN (SYNCHRIN,1);
    OWNER := LOCRINOWNER (WHICHCHILD);     CHILD1 IS SUSPENDED
       .                                     UNTIL ACTIVATED
       .                                        BY PARENT
     CHILDCOUNT := CHILDCOUNT + 1;
     ACTIVATE (OWNER);
    END; {WHILE LOOP}                              .
       .                                UNLOCKLOCRIN (WHICHCHILD);
       .                                           .
       .                                           .
```

Figure 2-5.

Example 2-5. Identifying a Local RIN Locker.

Both processes in Figure 2-5 share the following constants:

WHICHCHILD = 1	All processes in the program example agree that WHICHCHILD is the local RIN used to determine the identity of the child process that activated the parent process PARENT.
SYNCHRIN = 2	All processes in the program example agree that SYNCHRIN is the local RIN used to ensure that PARENT can execute required code located in the WHILE loop before another child process can activate it.
PARENTWAIT = 1	Used by both SUSPEND and ACTIVATE to indicate that the suspended process permits only its parent process to activate it with an ACTIVATE intrinsic call.
CHILDWAIT = 2	Used by both SUSPEND and ACTIVATE to indicate that the suspended process permits only a child process to activate it with an ACTIVATE intrinsic call.

In Example 2-5, PARENT waits in a suspended state, unable to execute until an activation signal is received from a child process (in this case, by CHILD1 calling ACTIVATE). Once activated, PARENT locks SYNCHRIN to synchronize its communication with CHILD1. LOCRINOWNER determines the identity of the process that activated PARENT (the process that locked WHICHCHILD). PARENT then performs its required duty within the WHILE loop.

PARENT activates the process that activated PARENT (CHILD1) then suspends itself to again await activation by a child process. Note that SYNCHRIN is passed as a parameter to SUSPEND. SUSPEND releases the RIN, making SYNCHRIN available to be locked by other processes.

For details on using SUSPEND and ACTIVATE, refer to the discussion of suspending and activating processes in *Process Management Programmer's Guide* (32650-90023).

Deadlock Considerations

If you are locking more than one RIN at a time in your program, there is a chance that you can cause a deadlock between two or more processes. Deadlock occurs when two or more processes are mutually blocked, waiting for each other to release a needed resource.

Example 2-6 illustrates how a deadlock can occur between two processes, Process A and Process B. Assume that both processes in the example are executing concurrently, line by line.

```
  {    PROCESS A EXECUTING     }       {    PROCESS B EXECUTING     }
                .                                    .
                .                                    .
                .                                    .
        LOCKGLORIN (1);                      LOCKGLORIN (2);
        LOCKGLORIN (2);                      LOCKGLORIN (1);

    PROCESS A REMAINS BLOCKED      PROCESS B REMAINS BLOCKED
    UNTIL RIN 2 IS UNLOCKED        UNTIL RIN 1 IS UNLOCKED
    BY PROCESS B. PROCESS A        BY PROCESS A. PROCESS B
    CANNOT UNLOCK RIN 1 WHILE      CANNOT UNLOCK RIN 2 WHILE
    BEING BLOCKED                  BEING BLOCKED
```

Figure 2-6.

Example 2-6. Deadlock.

In Example 2-6, Process A successfully locks global RIN 1, then attempts to lock global RIN 2 (already locked by Process B). Process A is blocked until it can successfully lock global RIN 2. While Process A is blocked, it cannot unlock global RIN 1, thus making global RIN 1 unavailable for locking by Process B.

Process B, meanwhile, has locked global RIN 2 and has been blocked attempting to lock global RIN 1 (locked, or course, by Process A). Global RIN 2 remains unavailable to Process A.

Both Process A and Process B find themselves mutually blocked and in the state of deadlock. Even if subsequent code in both program files unlocks one or both global RINs, neither process can execute that code.

One way to avoid deadlocks is by ranking the RINs used by cooperating processes. In Example 2-5, cooperating processes agree to first attempt to lock RIN 1 before attempting to lock RIN2. In addition, the processes that have successfully locked both RINs agree to unlock the two RINs in the reverse order, first RIN 2, then RIN 1. Because cooperating processes must lock RINs in ascending order and unlock them in descending order, deadlock cannot occur.

If you have only standard user capabilities and deadlock occurs between two or more processes in your program's process structure, you must abort your program to resolve the deadlock.

If you have Multiple RIN (MR) Capability and deadlock occurs between your program and processes located in different jobs/sessions, you must immediately contact your System Manager to resolve the deadlock.

3

Dynamic Loading of Library Procedures

Externally referenced procedures located in executable library files (XLs) are normally bound to your program when it is first loaded at process creation (load time). MPE XL enables you to bind and load a specified XL procedure any time during process execution (run time), a feature referred to as dynamic loading.

You might, for example, decide to do this for a large procedure used optionally and infrequently by your program, or for a procedure whose name is not known by your program at load time.

Introduction to HPGETPROC-PLABEL

The `HPGETPROCPLABEL` intrinsic dynamically loads an XL procedure and returns the procedure's label (plabel) to your program.

You can use the plabel to make a dynamic call to the procedure if the programming language contains features for making dynamic procedure calls, for example, the `CALL` procedure in HP Pascal/XL. Refer to the *HP Pascal Reference Manual* (31502-90001) for details on the `CALL` procedure.

The syntax of `HPGETPROCPLABEL` is as follows:

`HPGETPROCPLABEL(`*procname,plabel,status,firstfile,casesensitive*`)`

`HPGETPROCPLABEL` searches through a list of XLs (referred to as the binding sequence, described below) to locate the procedure you specify in the *procname* parameter. When the procedure is found, it is dynamically loaded and its plabel is returned to your program in the *plabel* parameter.

Appendix B contains a program example illustrating the use of the `HPGETPROCPLABEL` intrinsic.

The *status* parameter returns status information about the intrinsic call, in the following manner:

Bits (0:16)	When the signed integer value represented by these bits is zero, a normal status is indicated. A negative value indicates an error condition, and a positive value indicates a warning condition.

Bits (16:16)	The signed integer value represented by these bits defines the subsystem that set the status information in Bits (0:16). The NM loader identification number is 104. When the value represented by these bits is zero, a normal status is indicated.

Determining the Binding Sequence

The binding sequence is a list of XLs the loader searches to satisfy your program's unresolved external references. The loader creates the binding sequence at load time. The program file is placed first in the binding sequence. Any additional XLs are placed after the program file in the order in which you specified them. The System Libraries (`XL.PUB.SYS` followed by `NL.PUB.SYS`) are always placed last in the binding sequence. The order of the binding sequence is important, as the loader makes a single pass in one direction through the list.

MPE XL enables you to specify additional XLs you want placed in the binding sequence by using any of the methods described in groups one through three below.

An XL list specified using methods described in group two always override lists specified using methods described in group one. Likewise, lists specified using methods described in group three always override lists specified using methods described in groups one and two.

1. You can specify a list of your own executable libraries in the `XL=` parameter of the `:LINK` command.
2. You can specify a list of MPE XL-defined libraries in either the `LIB=` parameter of the `:RUN` command or the *flags* parameter LIBSEARCH bits of the `CREATE` or `CREATEPROCESS` intrinsics.
1. You can specify a list of your own executable libraries in the `XL=` parameter of the `:RUN` command, or Item Number 19 of the `CREATEPROCESS` intrinsic.

For example, if the calling process was created with the following command:

```
:RUN PROGRAM1; XL=LIBA,LIBB,LIBC,LIBD
```

The binding sequence illustrated in Figure 3-1 is created (arrows determine the search direction):

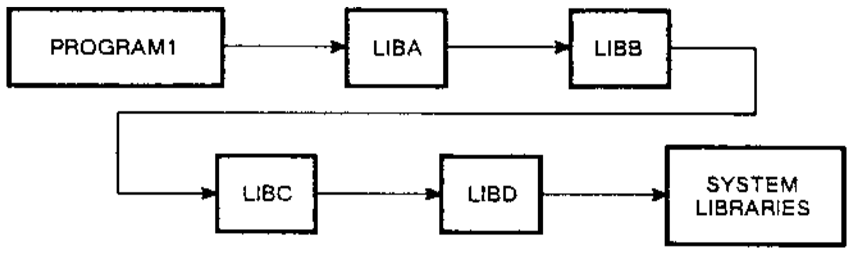


Figure 3-1. Illustration of a Load Time Binding Sequence.

(Refer to the *HP Link Editor/XL Reference Manual* (32650-90030) for more information about creating and maintaining XLs.)

Searching an Executable Library Not Specified at Load Time

HPGETPROCPLABEL can search for *procname* in one XL not specified at load time. The XL must be specified in *firstfile*, and any unresolved external references within the XL must be resolved only in the System Libraries. If *firstfile* is not found in the libraries available to the process, the XL is placed in a binding sequence independent of the original binding sequence prior to the dynamic loading of *procname*. The dynamically added library contains supplemental code that can only be executed via a dynamic procedure call.

Figure 3-2 shows a program loading using two user XLs and the System Library. Also, the process has dynamically loaded a procedure located in an XL not specified at load time. The original binding sequence of the process is not altered.

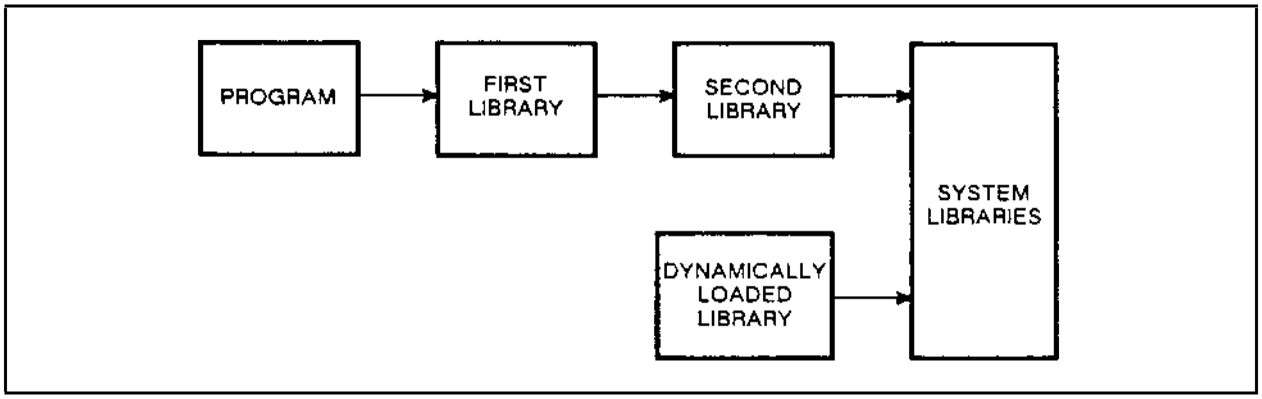


Figure 3-2. Illustration of Run Time Binding of an XL.

Searching Executable Libraries Specified at Load Time

You can use the optional *firstfile* parameter to specify which XL in the binding sequence you want HPGETPROCPLABEL to begin searching for *procname*. If you do not specify *firstfile*, only the the System Libraries are searched.

During a dynamic load, HPGETPROCPLABEL directs the loader to search through each XL in the binding sequence, beginning with the XL specified in *firstfile*, for the first instance of a procedure with the name *procname*. Each XL in the binding sequence after *firstfile* is searched in turn if:

- The procedure specified in *procname* is not located in the XL.
- The procedure specified in *procname* contains unresolved external references that need to be satisfied in subsequent XLs.

Note The same procedure name might be repeated in different XLs. To ensure that the loader locates the correct procedure, you must be certain that any XLs searched prior to the XL containing the correct procedure do not contain a different procedure with the same name.

For example, let us assume that the binding sequence in Figure 3-1 is that of your process executing `PROGRAM1`. Let us further assume that there is a procedure named `MYPROC` located in the XL named `LIBA` and a different procedure with the same name (`MYPROC`) located in the XL named `LIBC`.

The following `HPGETPROCPLABEL` call dynamically loads the `MYPROC` procedure located in `LIBC`:

```
              .
              .
              .
        PROCNAME := '%MYPROC%';
        FIRSTFILE := '%LIBB%';
        HPGETPROCPLABEL (PROCNAME,PLABEL,STATUS,FIRSTFILE);
              .
              .
              .
```

In the above example, the intrinsic directs the loader to search the portion of the binding sequence beginning with `LIBB` for the first instance of a procedure named `MYPROC. (LIBA` is never searched.) When `MYPROC` is not found in `LIBB`, the loader continues the search in `LIBC` (the next XL in the binding sequence). Once `MYPROC` is found in `LIBC` it is dynamically loaded and its plabel returned in `PLABEL`. The same result is accomplished if you specify `LIBC` in `FIRSTFILE`.

The following three intrinsics can return to your program the fully qualified file name of a particular XL in the binding sequence you want to specify in *firstfile*.

Using HPFIRSTLIBRARY

The `HPFIRSTLIBRARY` intrinsic returns the fully qualified file name of the first XL in the binding sequence (after the program file) determined at load time. If you did not specify additional XLs using one of the methods described, in "Determining the Binding Sequence", then the name of the first file in the System Libraries is returned. You can pass this name to `HPGETPROCPLABEL` in the *firstfile* parameter.

Using Figure 3-1 as your program's binding sequence, the following intrinsic call:

```
        HPFIRSTLIBRARY (XLNAME);
```

returns the fully qualified file name of the first XL in the binding sequence (after the program file), `LIBA.GROUP.ACCOUNT`.

Using HPMYPROGRAM

The `HPMYPROGRAM` intrinsic returns the fully qualified file name of the program being executed by the calling process (the first file in the binding sequence). You can pass this name to `HPGETPROCPLABEL` in the *firstfile*

parameter.

Using Figure 3-1 as the binding sequence for your program, the following call:

```
     HPMYPROGRAM (PROGNAME);
```

returns the fully qualified file name of your program file, `PROGRAM1.GROUP.ACCOUNT`.

Using HPMYFILE

The `HPMYFILE` intrinsic returns the fully qualified file name of the program or XL that called the intrinsic. If this intrinsic is called from your program, your program's file name is returned. If this intrinsic is called from a procedure located in an XL, the file name of the XL is returned. You can pass this name to `HPGETPROCPLABEL` in the *firstfile* parameter.

A

Global RIN Program Example

This HP Pascal/XL program illustrates how you can use the two intrinsics, `LOCKGLORIN` and `UNLOCKGLORIN`, to prevent simultaneous access to a selected record in a shared file while one user is updating the record. Five global RINs were previously acquired through the `:GETRIN` command. Each RIN is associated with a subset of 4 records in a 20 record data file. This method of assigning RINs allows other users to concurrently access other subsets of records in the same file. RIN-locking occurs in procedure `access_record_exclusively`. This program is intended to be used with the file `BOOKFILE` (illustrated in Chapter 2).

```
program global_RIN_example;
{*******************************************************************}
{                        DECLARATION PART                            }
{*******************************************************************}
const
  rinbase        = 1;                   {Lowest RIN assigned         }
  recds_per_rin  = 4;                   {Partition the datafile      }
  maxrin         = 5;                   {Highest RIN assigned        }
  CCG            = 0;                   {Condition Code Warning      }
  CCL            = 1;                   {Condition Code Error        }
  CCE            = 2;                   {Condition Code successful   }
  maxbooks       =19;                   {Last record in datafile     }

type                                    {holds titles and locations  }
  record_field   = packed array [1..36] of char;
                                        {record structure of datafile }
  library_record = packed record
                     title: record_field;     {Holds book title      }
                     location: record_field;  {Holds book location   }
                   end;

  hp_status = packed record
              case integer of
              0: (all:integer);
              1: (info:shortint;        {Error number from subsys    }
                  subsys: shortint);    {Subsystem number            }
                end;
var
  stdin,stdlist,booklist: integer;      {HPFOPEN file numbers        }
  ascii,perm,rw,share,cctl:integer;     {HPFOPEN item variables      }
  status: hp_status;                    {HPFOPEN intrinsic status    }
  length,accno,rin: shortint;           {Vars required by intrinsics  }
  lockflag: 0..65565;                   {Required by lock intrinsics  }
  bookrecord: library_record;           {Used in read/write operations}
  dummy: boolean;                       {Required by FCONTROL        }
  infile,outfile,datafile,             {File names used with HPFOPEN }
                                       {Required by LOCKGLORIN       }
  rinpassword: packed array [1..12] of char;
                                       {vars required by intrinsics  }
  buffer,change,head,request: record_field;

procedure hpfopen;        intrinsic;    {Opens three files            }
function fread:shortint;  intrinsic;    {Reads from $STDIN            }
procedure fwrite;         intrinsic;    {Writes to $STDLIST           }
procedure fcontrol;       intrinsic;    {Post to disc                 }
procedure freaddir;       intrinsic;    {Random reads from datafile   }
procedure fwritedir;      intrinsic;    {Random writes to datafile    }
procedure lockglorin;     intrinsic;    {RIN-locking intrinsic        }
procedure unlockglorin;   intrinsic;    {RIN-unlocking intrinsic      }
function binary:shortint; intrinsic;    {Convert ASCII to binary      }
procedure printfileinfo;  intrinsic;    {Used in Error Handler        }
procedure quit;           intrinsic;    {Used in Error Handler        }

procedure error_handler(filenum,quitnum: shortint);
{*******************************************************************}
{ procedure error_handler is invoked whenever a system intrinsic    }
{ call is unsuccessful.                                             }
{*******************************************************************}
  begin
                                       {If valid file number, then   }
                                       {print file info to $STDLIST  }
  if filenum >=0 then printfileinfo(filenum);
  quit(quitnum);                       {Abort process                }
  end;

procedure initialize_variables;
{*******************************************************************}
{ procedure initialize_variables initializes all global variables   }
{ prior to use.                                                     }
{*******************************************************************}
  begin
  infile:=  ' $stdin ';                 {associated with $STDIN      }
  outfile:= ' $stdlist ';               {associated with $STDLIST    }
  datafile:= ' bookfile ';              {formaldesignator = BOOKFILE }
  lockflag:= 1;
  rinpassword:= 'bookrin ';
  dummy:= true;
  status.all:= 0;
  ascii := 1;                          {ascii/binary option ASCII    }
  perm  := 1;                          {domain option PERMANENT      }
  rw    := 4;                          {access type option READ/WRITE}
  share := 3;                          {exclusive option SHARE       }
  cctl  := 1;                          {carriage control option CCTL }
  stdin := 0;
  stdlist := 0;
  booklist := 0;
  head:= 'LIBRARY INFORMATION PROGRAM ';  {Header introduces program  }
  change:= 'NEW LOCATION  ';              {User interface             }
  request:= 'ACCESSION NO: ';             {User interface             }
  end;

procedure open_files;
{*********************************************************************}
{ procedure open_files opens all files used by program.               }
{*********************************************************************}
  begin
  hpfopen(stdin,status,2,infile,3,perm,53,ascii);     {Open $STDIN    }
  if status.all <> 0 then error_handler(-1, status.info);
  hpfopen(stdlist,status,2,outfile,3,perm,
          7,cctl,53,ascii);                           {Open $STDLIST  }
  if status.all <> 0 then error_handler(-1, status.info);
  hpfopen(booklist,status,2,datafile,
          3,perm,53,ascii,11,rw,13,share);            {Open datafile  }
  if status.all <> 0 then error_handler(-1, status.info);
  end;

procedure select_record(var record_length: shortint;
                        var book_number: record_field);
{*********************************************************************}
{ procedure select_record allows user to select the bookrecord for    }
{ viewing and updating.                                               }
{*********************************************************************}
  begin
  fwrite(stdlist,request,7,208);        {Ask user for Book number     }
  if ccode <> CCE then error_handler(stdlist,101);
  record_length:= fread(stdin,buffer ,-10);   {Read user input        }
  if ccode <> CCE then error_handler(stdin,102);
  end;

procedure update_record;
{*********************************************************************}
{ procedure update_record prints the selected book record to $STDLIST,}
{ prompts user for new location, then reads the input from $STDIN.  If }
{ user supplies a location, record is updated, then posted to disc.   }
{*********************************************************************}
  begin
  fwrite(stdlist,bookrecord,-72,0);     {Print selected  bookrecord   }
  if ccode <> CCE then error_handler(stdlist,105);
  fwrite(stdlist,change,-14,208);       {Prompt user for new location }
  if ccode <> CCE then error_handler(stdlist,106);
  buffer:= '                                   '; {Clear variable     }
  length:= fread(stdin,buffer,-36);     {Read user-input new location }
  if ccode <> CCE then error_handler(stdin,107);
                                        {If user input characters,    }
                                        {update record in datafile    }
  if length > 0 then
    begin
    bookrecord.location:= buffer;         {Update location field      }
    fwritedir(booklist,bookrecord,-72,accno);   {Update datafile      }
    if ccode <> CCE then error_handler(booklist,108);
    fcontrol(booklist,2,dummy);           {Force posting to disc      }
    if ccode <> CCE then error_handler (booklist,109);
    end;
  end;

procedure access_record_exclusively(rinnum:shortint);
{*******************************************************************}
{ procedure access_record_exclusively locks the global rin associated }
{ with the selected bookrecord.  While the RIN is locked, others      }
{ attempting to lock the same RIN are denied.  While RIN is locked,   }
{ the user-selected  book record is read from the datafile, then     }
{ PROCEDURE update_record is invoked to update the location field of  }
{ the bookrecord.  After successful update, RIN is unlocked.          }
{*******************************************************************}
  begin
  lockglorin(rinnum,lockflag,rinpassword);  {Lock global RIN          }
  if ccode <> CCE then error_handler(-1,103);
  freaddir(booklist,bookrecord,-72,accno);  {Read selected bookrecord }
  if ccode = CCL then error_handler(booklist,104) else
  if ccode = CCE then update_record;    {Call PROCEDURE update_record }
  unlockglorin(rinnum);                 {Unlock global RIN            }
  if ccode <> CCE then error_handler(-1, 110);
  end;

procedure update_book_information;
{*******************************************************************}
{ procedure update_book_information is the main outer-block procedure.}
{*******************************************************************}
  begin
  fwrite(stdlist,head,14,0);       {Print program intro to $STDLIST   }
  if ccode <> CCE then error_handler(stdlist,4);
  select_record(length,buffer);    {Call record selection procedure   }
  while length <> 0 do
                                   {Continue loop so long as user     }
                                   {selects a bookrecord to update.   }
    begin
    accno:= binary(buffer,length);    {Converts ascii to shortint     }
    if ccode <> CCE then error_handler(-1,112) else
      begin                        {If accno is successfully converted,}
                                   {use it to compute RIN.            }
      rin:= rinbase + (accno div recds_per_rin);
                                   {If computed RIN one of those from }
                                   {:GETRIN, call PROCEDURE to access }
                                   {the selected record exclusively.  }
      if rin in [rinbase..maxrin]
      then access_record_exclusively(rin);
      end;
    select_record(length,buffer);  {Select another record, loop       }
    end;                                {Loop                              }
  end;
{**********************************************************************}
{                           MAIN PROGRAM PART                          }
{**********************************************************************}
begin
initialize_variables;
open_files;
update_book_information;
end.
```

Dynamic Loading Program Example

This HP Pascal/XL program example illustrates the use of the HPGETPROCPLABEL intrinsic.

HPGETPROCPLABEL returns the plabel of the COMMAND intrinsic. The program requests the user to input the name of a CI command. Both the plabel and the CI command are passed to the HP Pascal/XL CALL procedure which calls COMMAND and directs it to execute the user-specified CI command.

```
$standard_level 'FULL_MODCAL'$
$type_coercion 'NONCOMPATIBLE'$
$tables off$
PROGRAM hpgetprocplabel_test(input, output);

TYPE
   c80 = packed array [1..80] of char;
  mpexl_status = record
      case integer of
         0 : (all : integer);
         1 : (info   : shortint;
              subsys : shortint);
         end;
   Proc_Type = procedure(VAR command : c80; VAR error, parm : shortint);
   Proc_Name_Type = string[32];
   File_Name_Type = packed array [1..36] of char;
   bit32 = minint..maxint;
{  bit32 = 0..4294967295; }

VAR
   status      : mpexl_status;
   Proc_Name   : Proc_Name_Type;
   plabel      : bit32;
   Invoke_Proc : Proc_Type;
   command_str : string[80];
   command     : c80;
   str_cr      : string[1];
   cr          : packed array [1..1] of char;
   error, parm : shortint;
   cicat       : shortint;
   catname     : File_Name_Type;

PROCEDURE hpgetprocplabel;     INTRINSIC;
PROCEDURE fclose;              INTRINSIC;
FUNCTION  fopen : shortint;    INTRINSIC;
PROCEDURE genmessage;          INTRINSIC;
PROCEDURE terminate;           INTRINSIC;

begin
Proc_Name := ' COMMAND ';
writeln('About to call hpgetprocplabel');

hpgetprocplabel(Proc_Name, plabel, status);

if (status.all <> 0) or
   (ccode <> 2) then
   begin
   writeln('status.subsys:', status.subsys);
   writeln('status.info:  ', status.info);
   end
else
   begin
   writeln('hpgetprocplabel successfully called');
   end;

Invoke_Proc := Proc_Type(plabel); { coerce 32 bit ptr to 64 bit }

prompt('Enter MPE XL Command:');
readln(command_str);
cr[1] := chr(13);
strmove(1, cr, 1, str_cr, 1);
strappend(command_str, str_cr);
strmove(strlen(command_str), command_str, 1, command, 1);

CALL(Invoke_Proc, command, error, parm);

if error <> 0 then
   begin
   writeln('Error in command:', command);
   catname := 'catalog.pub.sys ';
   cicat   := fopen(catname, 5, octal('420'));
   genmessage(cicat, 2, abs(error));
   fclose(cicat, 0, 0);
   end;

end.
```

Index