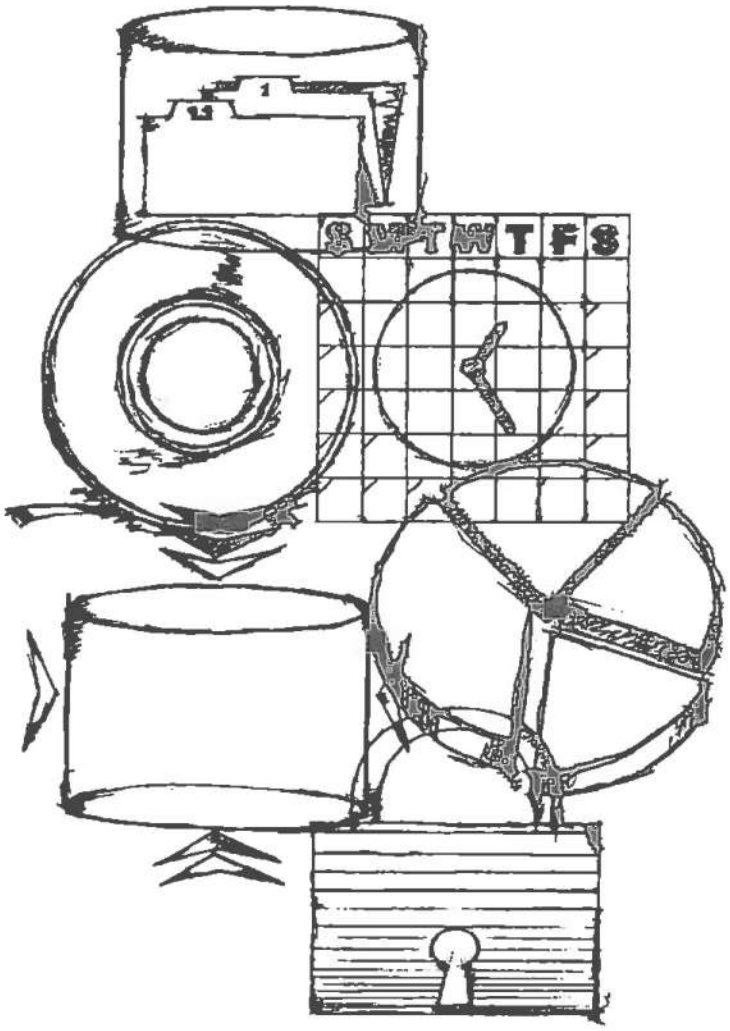
LIBRARIAN/iX™

User's Guide

Version 4.00
May 1998



OCS

Quality • Innovation • Service

LIBRARIAN/iX User's Guide
Version 4.00

Table of Contents

Chapter 3: File Transactions

Chapter 4: Revisions

Chapter 5: Printing, Scanning, and Comparing Files

Chapter 6: User Filesets

Chapter 7: Listfiles

Chapter 8: Rebuilding Applications with MAKE

List of Figures

List of Tables

Preface

Purpose of This Manual

The *LIBRARIAN/iX User's Guide* describes how to use LIBRARIAN. It is the companion piece to the *LIBRARIAN/iX Reference Guide* and *LIBRARIAN/iX Administrator's Guide.*

Audience

This manual is written for personnel who use LIBRARIAN on a daily basis, such as programmers, operators, and managers. Knowledge of basic operating system concepts and terminology is assumed. No previous knowledge of LIBRARIAN is required.

How This Manual is Organized

The *LIBRARIAN/iX Administrator's Guide* chapters are organized as follows:

Chapter 1	"Introduction": what LIBRARIAN does, and how it fits in to the application development cycle.
Chapter 2	"Getting Started": applying the Shortcut program to get started using LIBRARIAN.
Chapter 3	"File Transactions": how to move and copy files using steps, perform other file activities, and review information about files.
Chapter 4	"Revisions": how to branch from one version and how to merge two revisions.
Chapter 5	"Printing, Scanning, and Comparing Files": how to print, view, and edit files, show the file differentiators, and scan and replace strings of text.
Chapter 6	"User Filesets": creating and maintaining user filesets using **FMAINT** commands.
Chapter 7	"Listfiles": creating and using listfiles with **LMAINT** commands.
Chapter 8	"Rebuilding Applications with MAKE": how to rebuild applications with the **MAKE** facility.
Chapter 9	"Macros": how to create and use macros and procedure files.
Appendix A	"Applications in Progress": how to implement LIBRARIAN for applications with work already in progress.

Glossary	A Glossary of Terms is provided at the back of this guide as well as the index to the guide
Index	An index of LIBRARIAN topics at the end of this guide.

Conventions

We use the following conventions throughout this guide.

COMMANDS	All commands appear in bold capital letters. If a command can be abbreviated, the optional portion of the command is enclosed in brackets ([ ]). A blank space *must* separate the command from the parameter list.
KEYWORDS	Keywords and parameters (shown in bold capital letters) must be entered exactly as specified.
italics	Words or characters in italics represent variables or arguments that you must replace with an actual value. In the following example, you must replace *fileset* with the name of the file you want to copy. >COPY *fileset* Italics are also used to introduce new terminology or for emphasis.
punctuation	Enter punctuation exactly as shown. (Refer to specific instructions for brackets and braces, below.)
{ }	Braces enclose required elements. When there are several elements within braces, you must select one element. In the following example, you must select one of **PROCEDURES**, **PROJECTS**, or **STEPS**. >HELP { PROCEDURES | PROJECTS | STEPS }
[ ]	Brackets enclose optional elements. In the following example, brackets around the letters UPDATE indicate that you do not have to type the entire word. >AUTO[UPDATE]

If there are several elements, you can select any one or none of them. In the following example you can select **BATCH**, **CONFIRM** or **MEMO**, or none.

```
>COMPRESS [ filelist ]
    [ ;BATCH ]
    [ ;CONFIRM ]
    [ ;MEMO ]
```

When brackets are used, you cannot enter a value in the inner brackets unless you enter a value (wildcard or literal) in the outer brackets.

... An ellipsis indicates that the previous bracketed element can be repeated or that elements have been omitted.

& An ampersand indicates that the command continues on the next line.

The white flag symbol indicates that the text pertains to LIBRARIAN running under the MPE operating system.

The gray flag symbol indicates that the text pertains to LIBRARIAN running under the UNIX operating system.

The striped flag symbol indicates that the feature being described is only available with LIBRARIAN/iX–Plus.

MPE only

This symbol identifies LIBRARIAN commands that have no equivalent under the UNIX operating system.

File Naming Conventions

In specifying files, LIBRARIAN commands use the following wildcard conventions:

@ Zero or more alphabetic and/or numeric characters. Used alone, denotes all members of a set.

* Zero or more alphabetic and/or numeric characters. Used alone, denotes all members of a set.

Single numeric character.

? Single alphabetic or numeric character.

In addition, a slash (/), a single period and slash (./), a double period and slash (../), or a tilde and a slash (~/) immediately preceding a filename indicate a UNIX file.

Related Documentation

Along with this manual, you can refer to the following documentation by OCS.

The *LIBRARIAN/iX Reference Guide* provides information on LIBRARIAN functions, including complete command syntax and reference material for all LIBRARIAN features.

The *LIBRARIAN/iX Administrator's Guide* contains information on how to setup and maintain LIBRARIAN.

Online help contains the contents of all LIBRARIAN manuals. You can access online help with the **HELP** command or pressing **F1 (Help)** in menu mode.

Client Services

LIBRARIAN is supported by OCS Client Services, which is dedicated to providing timely and accurate information and solutions. For fast, accurate answers, we maintain a telephone hotline that includes emergency after-hours service. You can count on OCS to isolate any problems quickly and provide conscientious support and a fast response.

Operations Control Systems hotline numbers:
Phone (415) 493-4122
FAX (415) 493-3393

Your Comments

We value your comments. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. Please use the Reader's Comment Form at the end of this guide to tell us what you like and dislike about and of the OCS manuals.

Introduction 1

Your organization relies on the applications managed by your development team to stay in business. When an application fails in a production environment, your company loses time, productivity, and money. That is why controls to safeguard applications are so important. LIBRARIAN protects applications through the entire development cycle, from coding to production.

LIBRARIAN enforces a change control and testing discipline, and documents all changes to source code. LIBRARIAN centralizes access to source files to prevent simultaneous changes to the same code, synchronizes source versions with their related executable programs, and ensures that only authorized changes are incorporated into a production version. LIBRARIAN also controls and automates the move–to–production process, even across networks.

This chapter describes the functions of LIBRARIAN within the context of the application development cycle. Topics discussed include:

- Product Components and Concepts
- Master Library Management
- Change Control
- Configuration Management
- Automated Move–to–Production
- Audit Trails and Reporting
- LIBRARIAN/iX Plus Features
- Meeting Your Objectives with LIBRARIAN Features

Product Components and Concepts

LIBRARIAN offers a wide range of functionality in an easy–to–implement, easy–to–learn, and easy–to–use format. All LIBRARIAN functions can be accessed through on–screen, pull–down menus and context–sensitive online help is always available. LIBRARIAN is also easy to set up using the Shortcut utility described in Chapter 2, "Getting Started with Basic Rules" in the *LIBRARIAN/iX Administrator's Guide*. As you become familiar with LIBRARIAN functions and commands, you might prefer to use the command–line interface. You can easily switch between the menu and command–line interfaces to meet your needs and preferences.

LIBRARIAN consists of:

- the main LIBRARIAN program
- maintenance screens to set up and maintain rule definitions
- report programs
- utility programs for file housekeeping and mass changes
- databases to store rules, file tracking information, and audit information
- a MAKE utility to rebuild applications and synchronize libraries
- delta and generation files containing file revisions

Master Library Management

Because LIBRARIAN is designed to automate and manage functions relating to the application development cycle, the *application* provides an organizational framework for file management activities. The files you manage with LIBRARIAN are called *master files*. You include the master files for an application in a *master library* and define how the files can be accessed, copied, and/or replaced. Copies of master files in other locations are called *secondary files* or *secondaries*. These copies can be development or maintenance work in progress, or copies for reference only. Within a master library, you can create a hierarchy of filesets to meet your particular file management needs.

One distinct advantage LIBRARIAN offers is the ability to define and group collections of files (for example, the files related to an application) and associated rules that govern your library organization, allowable file movements, and user authorizations. You can define different sets of rules for each application to match the needs of your environment. See Chapter 3 "Master Library"and Chapter 6, "Projects", in the *LIBRARIAN/iX Administrator's Guide*, and see Chapter 6, "User Filesets" in this manual, for more information.

Change Control

LIBRARIAN prevents duplicate updates, accidental deletions, wrong versions, and lost programs. The standards and procedures you define are automatically enforced because all file movements and authorized operations must be performed through LIBRARIAN. LIBRARIAN automates the entire file movement cycle, from the time files are checked out of the library, through maintenance, development, and distribution activities. All actions are logged to an audit trail database.

Your file movement rules reflect your own established procedures and define how and where copies of master files are made, how approvals are noted, and how master files are replaced. You define these rules as *steps* and *routes*. A step is a specific file movement and a route is a complete cycle of individual file movements, including checkpoints and prerequisites. Defining file movement rules is discussed in Chapter 6 of the *LIBRARIAN/iX Administrator's Guide*.

Configuration Management

LIBRARIAN allows you to define multiple baselines or *versions* for your applications at strategic points in time. You can easily recreate an application as it was at the time you created the baseline. LIBRARIAN manages both revisions to individual files in an application and changes to the entire set of files that make up versions of applications.

LIBRARIAN also allows you to *branch* from the main development path to support cases where, for example, you need to fix or send out a *patch* for a problem with a previous file revision currently in production. LIBRARIAN will allow you to force branching, for example, to support a situation where you need to work on a file that is being worked on by someone else, but do not want your changes to be reflected in the main development path. For more information on LIBRARIAN's configuration management capabilities, please see Chapter 7, "Versions", in the *LIBRARIAN/iX Administrator's Guide* and Chapter 4, "Revisions", in this manual.

Automated Move-to-Production

LIBRARIAN controls and automates the move–to–production process, even across networked, heterogeneous, and/or remote systems. Production moves can be scheduled during off–hours to accommodate online users, and can be configured with automatic recovery in case of an incomplete update. Refer to Chapter 2, "Getting Started with Basic Rules" in the *LIBRARIAN/iX Administrator's Guide* for more information.

Audit Trails and Reporting

LIBRARIAN eliminates the tedious manual task of documenting file changes and activities. All changes to master library files and all file movements and activities are automatically recorded in LIBRARIAN's audit trail database. You can also include memo text with transactions for documentation.

LIBRARIAN also offers reports and online inquiries to let you review the rules you have defined, file status and history information, and the audit trail records. This manual, the *LIBRARIAN/iX Administrator's Guide*, and the *LIBRARIAN/iX Reference Guide* each have chapters detailing LIBRARIAN's reporting functions.

LIBRARIAN/iX Plus Features

For teams that maintain extensive releases of software, maintain existing versions of applications while developing new releases, or have several developers working on the same source code simultaneously, OCS recommends LIBRARIAN/iX Plus. The LIBRARIAN/iX Plus package includes the standard LIBRARIAN features, plus:

- Delta Management
- Merge
- Annotated Source Listings
- LCOMPARE

Delta Management

Delta management cuts disk space overhead by keeping only the changes to a source file rather than saving complete revisions. To accomplish this, LIBRARIAN creates a special file that contains the original version of the source file and a history of all changes made for each subsequent revision to that file. Delta files provide the data necessary for annotation.

Merge

The **MERGE** option lets you combine source code changes from one or more branches to the main development path. For more information on branching, see Chapter 4, "Revisions" in this manual.

In a case where modifications to one file may need to be split among several programmers, a branch can be created for each programmer to work on individual tasks. The **MERGE** option lets you combine these different branches when the work is completed.

To protect your source code from conflicts that can occur when the same code is modified simultaneously by more than one programmer, the **MERGE** option highlights conflicting changes with comments indicating items that should be resolved prior to the next development step.

Another case where the **MERGE** function is needed is for patches to production releases. You can create a patch to fix the current production release of an application, and then **MERGE** these changes into the current development path for the application.

Source Code Annotation

Source Code Annotation creates a listing of source code showing lines that were inserted and deleted for each revision to the file, including date, time, user, and project information. **ANNOTATE** is an option of the LIBRARIAN **PRINT** and **COPY** commands and requires the use of the delta storage option.

LCOMPARE

The **LCOMPARE** command provides a quick and easy way to identify what has changed between two different copies, versions, or revisions of files. This can help eliminate common problems such as duplicate updates and accidental deletions and is a useful tool in the development, maintenance, and testing cycles.

Meeting Your Objectives with LIBRARIAN Features

LIBRARIAN provides a robust set of features that allow you to achieve a wide range of file management objectives. The following table matches typical file management objectives to the corresponding LIBRARIAN feature.

Table 1-1. LIBRARIAN Features Related to Objectives

File Management Objective	Corresponding LIBRARIAN Features
1. Improve efficiency and convenience	LIBRARIAN provides mass file movements, customized file movement commands, file push movements across accounts and system boundaries, complete audit trail, and automated maintenance.
2. Control copies of source, object, jobstreams, etc.	Define files in a master library.
3. Require approval of changes	Define CHECKIN step requiring an approval prestep. Authorize specific users to perform the APPROVE step.
4. Require testing of changes	Define rules requiring a step to document testing before allowing checkin.
5. Prevent overlapping changes	Assign serial access control to files.
6. Synchronize source/object	Use MAKE to compile changed source automatically, use VERIFY option on checkin and file distribution.
7. Enforce separation of duties	Authorize different users to perform specific steps.
8. Require independent testing	Authorize specific users to perform testing.
9. Restrict access to master files based on application	Authorize programmers only for steps within specific applications.
10. Associate work with project or service request	Require project codes for all steps in the route; authorize programmers for specific projects.
11. Maintain backup copies of old versions	Use "retention" feature on checkin.
12. Control versions on remote systems	Use LIBRARIAN to distribute software; audit trail tracks versions.
13. Provide audit trails	LIBRARIAN automatically maintains an audit trail of all file movements.
14. Review specific file changes	Use **S/COMPARE** to compare file versions and display differences.
15. Maintain current release while developing next release	Use Forward Versioning and separate maintenance and development routes.
16. Maintain concurrent revisions of individual programs	Use the revision control facility.
17. Tracking versions	Use LIBRARIAN version stamping facilities.

Getting Started 2

This chapter describes the basics of how to use LIBRARIAN. The following topics are covered in this chapter:

- How to Run LIBRARIAN
- Providing Your User ID and Password
- Changing Your Password and Lockword
- Switching to Another User ID
- Menu Mode
- Command Mode
- Shell Commands
- Online Help
- Performing Steps and Other File Activities

How to Run LIBRARIAN

To run LIBRARIAN from MPE, type:

```
:LIB
```

To run LIBRARIAN from UNIX, type

```
HP-UX[1] ocslib   if path set,
 otherwise

HP-UX[1] $OCSLIBDIR/ocslib
```

where $OCSLIBDIR is the name of the directory where the LIBRARIAN client software is installed.

Background Process on UNIX Clients

UNIX users can run a background process to issue LIBRARIAN commands from a UNIX shell prompt or within a script.

Since the background process maintains its connection to the server, LIBRARIAN is ready to accept requests at any time without the overhead of reconnecting. This capability greatly improves performance.

One use of the background feature is to check out files from MAKE files.

Start a background process by entering the following:

```
$ocslib  -bg
```

To issue LIBRARIAN commands to the background process, use the following syntax:

```
$ocslib  -fg  command
```

where command can be any of the LIBRARIAN commands, except screens and utilities. For security reasons, all requests must be made from the same terminal (or terminal window).

To terminate the background process, enter the exit command, as shown below:

```
$ocslib  -fg  exit
```

If your command includes delimiters or special characters that the shell might interpret, you must use a prefix of "\" with these characters, or enclose the entire command portion in quotations.

Note 

The background process inherits its environment from the process you started from, including the working directory and environment variables. However, you can change the current directory for the background process, as shown below:

```
$ocslib  -fg  cd  directory name
```

Providing Your User ID and Password

Before you can perform any LIBRARIAN functions, you must identify yourself with your LIBRARIAN user ID and personal password by responding to the prompts. Your response to the password prompt will not be displayed as you type it.

You will be prompted to assign a password the first time you use LIBRARIAN. You can change your own password at any time using the **PASSWORD** parameter of the LIBRARIAN **USER** command, or from the User menu as discussed below.

If you do not have a user ID, contact your LIBRARIAN Manager. The LIBRARIAN Manager also assigns any special user capabilities and specific LIBRARIAN step authorizations. As a general user, you can access and update your own user data on the Users (US) screen.

If your UNIX login user ID matches a LIBRARIAN user definition, then you can press RETURN to accept the default user at the user ID prompt. In this case, a password is not required.

Note 

All LIBRARIAN user IDs and passwords are case-sensitive.

Changing Your Password and Lockword

If you want to change your password or personal lockword (MPE), use the **PASSWORD** and/or **LOCKWORD** options of the **USER** command or select **Passwords** from the **User** menu. The following example shows how FRANK would change his password with the **USER** command:

```
>USER FRANK;PASSWORD
    New password?
    Please verify new password by typing it again
    New Password?
    User data successfully updated
```

If you want to remove your password or personal lockword, do not supply a new value. For example, remove the password for user name FRANK by typing:

```
>USER FRANK;PASSWORD=
```

Password Security Features

The LIBRARIAN Administrator can configure the following password security features:

- Aging (Days valid)
- Minimum length
- Maximum tries
- Disable user after maximum tries?

For more information, see the section "Setting Password Security" in the *LIBRARIAN Administrator's Guide*.

Switching to Another User ID

If you have more than one LIBRARIAN user ID assigned to you, you can switch your active user ID at any time by issuing the **USER** command or selecting **Identification** from the **User** menu. Issuing this command without parameters displays how you are currently signed on to LIBRARIAN.

Menu Mode

LIBRARIAN operates in menu mode by default. This mode allows you to execute commands and select options from a set of menus. The main menu consists of a horizontal menu bar that appears at the top of the screen under the OCS/LIBRARIAN title bar. The title bar shows whether you are running LIBRARIAN under MPE/iX or UNIX. (see Figure 2-1). To bypass the menus, see the section entitled "Command Mode" in the following section:

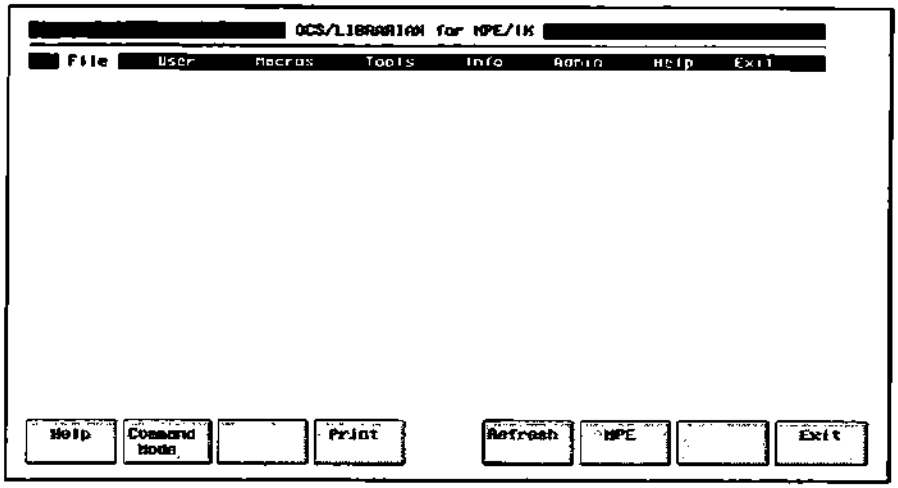


Figure 2-1. LIBRARIAN Main Menu

The title bar and main menu are displayed at the top of the screen while LIBRARIAN is running in menu mode. LIBRARIAN also provides a command line alternative discussed below. Function keys are defined as follows:

■ **F1** — Help	Gives context-sensitive help on the current menu option or dialog item.
■ **F2** — Command Mode/ Menu Mode	Toggles between the two modes.
■ **F3**	Not used.
■ **F4** — Print	Sends the current window to the printer.
■ **F5** — Refresh	Redraws the display.
■ **F6** — MPE/Shell	Exits temporarily to the MPE/iX or UNIX shell.
■ **F7**	Not used.
■ **F8** — Close	Exits LIBRARIAN. You can also select **Exit** from the Main menu bar or from the **File** menu.

Use the menus to issue the commands described in detail in the *LIBRARIAN/iX Reference Guide.*

Access and operate the menus as follows:

1. Use the right and left arrow keys to highlight the appropriate main menu option, or type the first letter of the menu option followed by RETURN.
2. Press RETURN.
3. Use the up and down arrow keys to highlight the desired pull-down menu item.
4. Press RETURN to select the item.
5. If applicable, enter the appropriate information in the dialog window and press Go (F7) or Cancel (F8).

Use the F8 function key to return to the main menu.

Warning

The Esc key should not be used in menu mode! This key signals that an escape sequence follows, and can cause unexpected behavior when you press additional keys.

Command Mode

As an alternative to menu mode and for batch mode operation, a command line interface is also provided. When you switch to command mode from menu mode, the screen clears and the > prompt is displayed.

You can switch between menu and command modes by pressing the F2 function key. You can also put the command **MENU OFF** in an **AUTOXEQ** macro file to bypass the menus automatically when you run LIBRARIAN. Macros are discussed in Chapter 9, "Macros", in this manual.

Shell Commands

In addition to LIBRARIAN commands, you can run any of the following from LIBRARIAN:

- MPE or UNIX commands
- UNIX scripts
- MPE UDCs
- MPE user programs

There are two ways to run any of the above, without exiting LIBRARIAN:

1. Type a colon (:) at the LIBRARIAN prompt >, and then press RETURN to break to the MPE/UNIX shell, or press **F6**.

   For MPE, you can only issue commands that are available in BREAK mode. Type **RESUME** to return to LIBRARIAN from MPE.

   Type **exit** to return to LIBRARIAN from the UNIX shell.

2. Issue a UNIX or MPE command preceded by a colon (:) at the LIBRARIAN prompt >. The colon is optional if it will not be confused with a LIBRARIAN command.

You can configure the LIBRARIAN prompt by setting a system variable called LIBPROMPT prior to running LIBRARIAN to the string you want to use as a prompt. For example:


```
:SETVAR LIBPROMPT "LIB>"
```


```
:export LIBPROMPT = "LIB>"
```

You can put the SETVAR statement in the LIBRARIAN UDC file for the **LIB** and **LIBSERV** commands on the MPE/iX server. Then, each time you run LIBRARIAN, the prompt is set automatically.

Online Help

Comprehensive online help is available for all LIBRARIAN commands and their parameters. Use the **F1** function key or the **HELP** command.

You can also select **Help** from the main menu bar to open the help index for help on a variety of topics, review the glossary, or get information about the current version of LIBRARIAN that you are running.

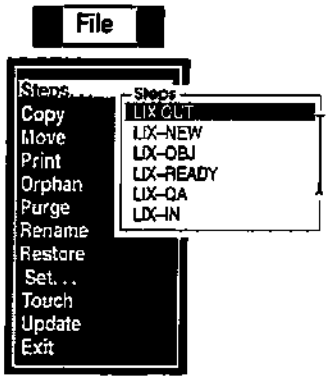
Performing Steps and Other File Activities

Once your LIBRARIAN Administrator has defined file movement rules to LIBRARIAN using Shortcut, you are ready to use LIBRARIAN to manage software files and control changes.

Each step moves or copies files based on rules defined in the database by the LIBRARIAN Manager, Application Manager or Rule Administrator. Each step definition, such as for checkout, identifies which files are valid, the destination location, any prerequisites, special operational rules, and default parameters.

Performing Steps in Menu Mode

If you use the menus, select **Steps** from the **File** menu. This menu includes only the steps you are authorized to perform.



The **File** menu displays a list of operations related to files.

Steps Displays a list of steps you are authorized to perform.

Step Dialog

When you select a step from the **Steps** menu, the dialog box shown in Figure 2–2 appears.

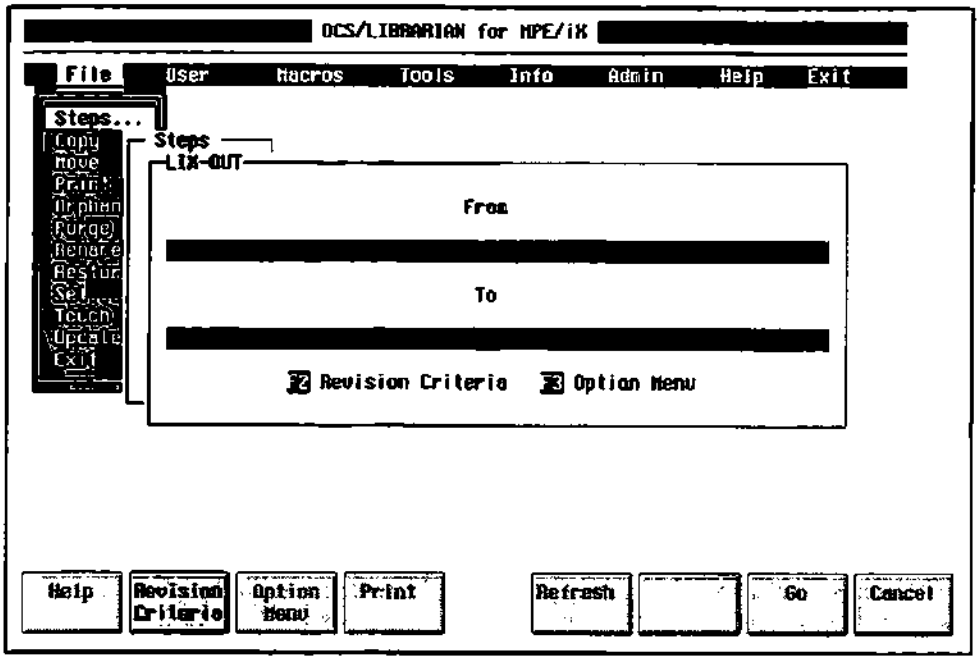


Figure 2–2. Step Dialog

In this dialog, you can enter source and destination file(s). Note that these fields scroll to the left if you type past the end of the field.

Enter the names of the files you want the step to process in the **From** field as described in "How to Refer to Files" in the next chapter. If you do not use absolute pathnames (fully qualified filenames), LIBRARIAN uses the step definition to determine the location of files. For other LIBRARIAN commands, LIBRARIAN uses your current working directory to locate files.

The **To** field is optional. LIBRARIAN uses the step definition to determine where to create files. You can only override this location if wildcards were used in defining the destination location for the step. If you leave this field blank for non-step operations, LIBRARIAN creates files in your current working directory.

You can apply revision criteria to the files listed by pressing **F2**. The **Revision Criteria** menu appears as shown in Figure 2-3 :

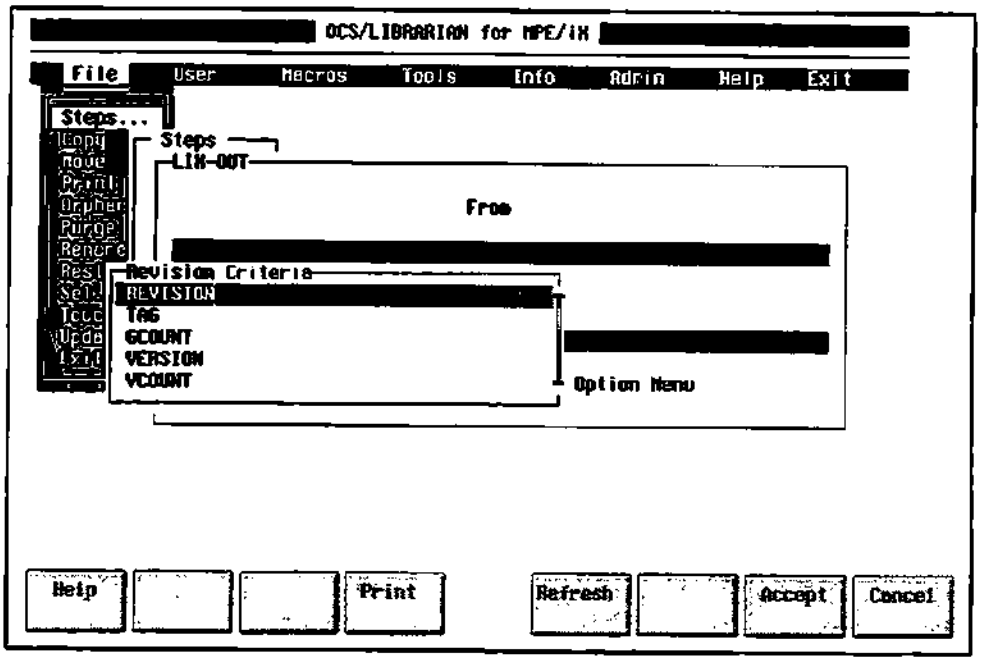


Figure 2-3. Revision Criteria Menu

When you select an option from this menu, a field appears allowing you to specify a value. Press **F8** (Cancel) to leave this menu without accepting the options you selected, or press **F7** (Apply) to leave this menu, applying the options you selected.

You can select step options and override default parameters by pressing **F3** (Option Menu) in the dialog box. A menu of the most common options appears as shown in Figure 2-4.

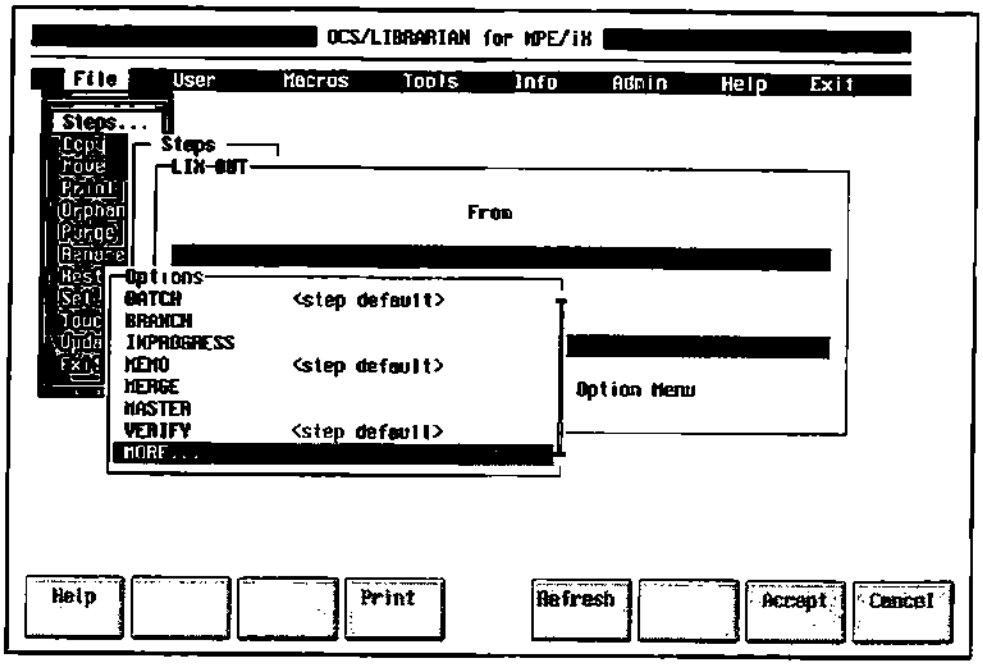


Figure 2-4. Step Options Menu

Other options are available by selecting **More...** from this menu. When you are finished selecting options and/or setting parameter values, press **F7** to accept the values or **F8** to cancel. You will return to the main step dialog box.

After specifying files, revision criteria and/or options, press **F7** (Go) to proceed with the transaction, or press **F8** (Cancel) to return to the previous menu.

Performing Steps from the Command Line

Perform a step from the command line by using the **PERFORM** command. For example, to perform the AP-OUT step you could type:

```
>PERFORM AP-OUT
```

You do not need to include **PERFORM** in the command. Instead, you can simply type:

```
>AP-OUT
```

If the step name used in this command is part of more than one route or application, LIBRARIAN displays a menu of steps, and you select the desired step from the menu.

If you know that there is more than one step with the same name and you want to identify the step uniquely when typing the command, use the full step name (step, route, and application). For example, the sample command above could also have been entered as:

```
>AP-OUT.APDEVEL.FIN
```

When you use the step name without parameters, LIBRARIAN automatically attempts to authorize all of the files identified in the step definition. In most cases you specify that the step should be performed for a subset of files. The next chapter describes how to specify subsets of files and additional parameters.

Common LIBRARIAN Commands

Table 2–1 lists some commonly used commands. These commands are also available from the menus. If you prefer to enter commands, use F2 to switch between menu mode and command mode.

Note

xxxx refers to the application name defined in Shortcut, up to four characters long. The > symbol is the standard LIBRARIAN prompt.

Table 2–1 Common LIBRARIAN Commands

Command	Purpose
>HELP STEPS	Displays information on which steps you can perform.
>xxxx-OUT *filename*	Check out file(s) from the master library to development location. A *violation* occurs if the checkout would replace another tracked file. A *conditional* read-mode checkout allowed if another user has checked out the file to another location.
>xxxx-NEW *filename*	Introduce new file(s) to an application. A "new" file is a new program or source file you have created, or a newly compiled object file not checked out. Introduce new files in the same group they will occupy in the master library.
>xxxx-OK *filename*	(Optional) Approve files to be moved out of development, either to a test location or back to the library.
>xxxx-TEST *filename*	(Optional) Move file(s) from development to a test location. Generally source and object are moved together to test. Move does not leave a copy in development.
>xxxx-FAIL *filename*	(Optional) Move file(s) back to development from test location. Generally both source and object are returned to development if executable code fails testing.
>xxxx-TESTOK *filename*	(Optional) Approve file(s) to be checked in from the test location to the master library. Must be done before the checkin.

Table 2-1 Common LIBRARIAN Commands (continued)

Command	Purpose
>xxxx-IN *filename*	Check in file(s) from test or development. Automatically retains (archives on disk) the old master file and compresses it.
>xxxx-READ *filename*	Copy file(s) from the master library to the development location in read mode. Read copies cannot be checked back in to the library. To clone a file, use this command with the **ORPHAN** option, rename the file, and use xxxx-NEW to introduce it to LIBRARIAN.
>PURGE *filename*	Delete a file that was checked out and is no longer needed. This command can also be used to remove your current copy in order to replace it with a fresh copy from the master library.
>VERIFY *filename*	Request information about a file or files. User is presented with a menu of formats showing available information ranging from file code and modification date to the file's LIBRARIAN owner or step history.

For details on command syntax and usage, refer to Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide*. Extensive online help is always available. To get help, use the **HELP** command or press **F1**.

File Transactions 3

You can perform all activities related to file movement using LIBRARIAN. In fact, your system administrator may have set up file system security in a way that only allows access to files through LIBRARIAN. LIBRARIAN authorizes files and performs single system or networked file operations based on rules stored in the rules database, and logs all transactions to an audit database.

This chapter describes how to move and copy files using steps, perform other file activities, and review information about files. Topics discussed in this chapter include:

- Overview of File Transactions
- How to Refer to Files
- How to Refer to Destinations
- How to Perform Steps
- Memos
- Using Personal Lockwords
- Macros
- Other File Operations
- Batch Transactions
- How to Check Transaction Status
- Reviewing File Information

Overview of File Transactions

You can perform all of your file movements from a single LIBRARIAN session. There is no need to log on to different accounts or directories to copy or move files. LIBRARIAN automatically pushes authorized files to the correct destination, across account and system boundaries.

Most file operations are done by executing steps defined by the LIBRARIAN Manager or Application Manager. You can perform these steps on the command line or select a step from the **Steps** menu which you can access from the **File** menu.

Each step definition identifies the part of the library (master fileset) to which the step applies, as well as the general location of valid files for the step. LIBRARIAN will only authorize files that are both members of the step's fileset and within the scope of the source location defined for the step.

There are many other LIBRARIAN operations available for files that LIBRARIAN is tracking. These operations also appear on the **File** menu and on the **Tools** menu.

In addition, the LIBRARIAN **X** commands allow you to operate on files that LIBRARIAN is not tracking. If you have **X** capability, file system security is ignored; otherwise, file system security is enforced.

How to Refer to Files

There are several ways you can refer to files when you perform steps or other LIBRARIAN commands. This section describes each of these methods.

Direct References

Filename

You can directly refer to files by name and location. The syntax for MPE is:

[*system:*] *file* [*.group* [*.acct*]]

where *file*, *group*, and *acct* identify the MPE filename. You can use wildcards consistent with MPE **LISTF** conventions. The syntax for UNIX is:

[*system:*] */*[*path.../*] *file*

where *file* identifies filename, including path preceding the filename. Use wildcards consistent with UNIX conventions (see "Filenaming Conventions" in the Preface of this guide).

For both MPE and UNIX, *system* is the name of the system where the file is located. Your current login values are used for omitted elements, except when performing steps, in which case configured values are used.

By default, LIBRARIAN treats all path references recursively; That is, all files in subdirectories of any directory specified are included when LIBRARIAN authorizes files. Recursion can be disabled by adding a suffix of a plus sign followed by a zero (+0) to the file reference.

For example, /usr/devel/d*+0 finds all files starting with the letter "d" in the devel directory without including files that are in subdirectories starting with the letter "d."

You can also control the number of levels of recursion, by adding a suffix of "+n", where "n" is the maximum number of directory levels to traverse.

Logical Fileset

You can directly refer to files in a logical fileset by specifying the fileset name preceded by a percent sign (%). A logical fileset can be a master fileset, user fileset, or project fileset. The syntax is:

%fileset

Listfile (Indirect File)

A listfile is a file that contains a list of filenames. You can use listfiles as a way to refer to files in all LIBRARIAN commands. Create listfiles using the LMAINT module of LIBRARIAN or with the editor of your choice. You can directly refer to files in a listfile by specifying the listfile name preceded by an exclamation point (!) or a caret (^). The syntax is:

!filename

Files from the Last Transaction

You can directly refer to files from the last logged transaction by specifying a star (*) or double–star (**). Destination files associated with, or the files processed in the last logged transaction, are authorized. The syntax is:

* (MPE) or ** (UNIX)

The single or double asterisk refers to the destination files successfully processed in the last transaction (or frozen with the **SET *** command).

Note

To use this feature, transaction logging must be enabled on the System Profile (SP) screen.

Indirect References

In a menu mode file dialog, press the **F2** function key (Revision Criteria) to specify indirect criteria.

Revision

You can indirectly refer to revisions of master files by specifying the master file(s) and a revision ID. The syntax is:

MPE
UNIX

{ [*system:*] *file* [.*group* [.*acct*]] | [*system:*] /[*path.../*] *file* | *%fileset* };REV[ISION]=*revision-id*|ALL

The revision ID is in the format VERSION:VCOUNT [.BRANCH.LEAF...].

You can authorize all revisions of a master file when using the **SET** and **PURGE** commands. To do this, use the **REVISION** parameter with the value of ALL. For example,

```
PURGE MYFILE.PUB.LIBRARY;REVISION=ALL
```

purges all revisions of the files associated with the master file, MYFILE.PUB.LIBRARY.

Version and Version Count

You can indirectly refer to versions of master files by specifying the master file(s) and a version and version count. The syntax is:

versionid OF { [*system:*] *file* [.*group* [.*acct*]] | [*system:*] /[*path.../*] *file* | *%fileset* } [;VCOUNT=*versioncount*]

Versionid is the identifier of a version. If the application for the version is ambiguous, LIBRARIAN prompts for it.

VCOUNT identifies the files with a version count equal to **VCOUNT** (the number of times the master file has been revised since the base version was created). Default: 0 (baseline version).

A **VCOUNT** value of **LAST** causes LIBRARIAN to operate on the last revision of a file within a version.

For example, the following command copies the latest revision of each file in the 1.0 version to the V100 area:

```
>COPY 1.0 OF %FINANCE TO =.=.V100;VCOUNT=LAST;OLDNAME
```

```
>COPY 1.0 OF %FINANCE TO /apps/gl/v100/(3,*);VCOUNT=LAST;OLDNAME
```

A **VCOUNT** value of **LASTNOT0** causes LIBRARIAN to operate on the last revision of a file within a retained version that is not a base revision. (e.g. to create a patch tape.)

For example, the following command distributes only those files that have changed since the base version was distributed:

```
>COPY RE.1.0 OF %MYFILES TO =.=.RELEASE;VCOUNT=LASTNOT0
```

```
>COPY RE.1.0 OF %MYFILES TO /apps/gl/release/(3,*);VCOUNT=LASTNOT0
```

Generation Count

You can indirectly refer to generations of a master file by specifying a master file(s) and a generation count. The syntax is:

{ [*system:*] *file* [.*group* [.*acct*]] | [*system:*] /[*path.../*] *file* | *%fileset* } ;GCOUNT = [-] *gcount*

The **GCOUNT** parameter directs LIBRARIAN to operate on files with the specified generation count (total number of times the master file has been replaced since its creation). This value can be either a positive or a negative value.

A negative value describes the generation relative to the current generation. For example, GCOUNT = -2 specifies files two generations prior to the current one.

Secondary Location

You can indirectly refer to secondary files by specifying the master filename(s) and the general location to search for associated secondaries. The syntax for MPE is:

```
{ %fileset                          AT }
{ [system:]file [.group [.acct ] ]  AT }  [system:]file [.group [.acct ]]
```

When using this syntax, LIBRARIAN operates on secondaries of the specified master files found in the specified secondary (AT) location. For example:

```
%AP-FILES AT @:@.@.@
```

refers to all secondary copies of %AP-FILES.

Alternatively, the syntax for UNIX is:

```
{ %fileset                      AT }
{ [system:] /[path.../] file    AT }  [system:] /[path.../] file
```

When using this syntax, LIBRARIAN operates on secondaries of the specified master files found in the specified secondary (AT) location. For example:

```
%AP-FILES AT *:/*
```

refers to all secondary copies of %AP-FILES.

Note 

In a menu mode dialog, enter the "AT" syntax directly in the filelist field, as you would in command mode.

Implied Reference by Project

You can imply the files associated with a project when performing a step by specifying the project name, rather than files. The syntax is:

```
>step.project
```

Alternatively, you can omit the project name and select your project from the project menu when projects are defined. In menu mode, this is the only alternative.

Implied Reference by Step

If you do not specify any files when performing a step, the step fileset (as defined on the Step (ST) screen) is used. For example:

>step

If projects are being used, you are presented a menu of projects. By selecting a project, you imply the project fileset when no files are specified.

Multiple File References

You can refer to multiple files combining any of the methods described above. Use commas to create a list of file specifications. The syntax is:

filelist [, *filelist* [, ...]]

Each filelist is processed by the LIBRARIAN program in a single transaction.

Exclusions Selection

This method designates files to be excluded from the operation. The syntax is:

{ – [*system:*] *file* [.*group* [.*acct*]] | – [*system:*] /[*path.../*] *file* | – *%fileset* }

When specifying multiple filelists, specify the exclusion(s) last. Exclusion(s) must be direct references, with or without wildcards. Use commas to separate exclusions.

Subset Selection

Project

Subset selection by project selects only files associated with a particular project. This parameter must follow all file references, including destination locations, if specified. **PROJECT** is valid for all commands. The syntax is:

filelist; PROJECT=*proj*

If you use a step to copy files in read–mode (e.g., move–to–production), LIBRARIAN automatically copies the appropriate revisions of the files associated with the project that you specify. However, if you do not use a step for file distribution (e.g., **COPY**), then use the project fileset as well as the **PROJECT** parameter.

Tag

Subset selection by tag selects only files that were assigned a specific tag with the **SET TAG** command. This parameter must follow all file references including destination locations, if specified. The syntax is:

filelist;TAG=*tagid*

Modification Status

Subset selection by modification status selects files based upon whether or not they have been modified since LIBRARIAN created them. Use the **MODIFIED** or **UNMODIFIED** parameters to select only those files modified or not modified since they were last copied or moved by LIBRARIAN. The current timestamp in the file label is compared with the timestamp in the LIBRARIAN database. The syntax is:

filelist;MODIFIED

User Confirmation

Subset selection by user confirmation has LIBRARIAN prompt for confirmation of each authorized file prior to processing. Use the **CONFIRM** parameter to request prompting. Files not confirmed are excluded from the operation. The syntax is:

filelist;CONFIRM

Tracking Status

Subset selection by tracking status lets you select files being tracked by LIBRARIAN, excluding those not being tracked. This applies only to ad hoc commands, such as **COPY** and **PURGE**. To include only untracked files, prefix these commands with **X**. The syntax is:

filelist;TRACKED

How to Refer to Destinations

Edit masks are used to determine the correct destination given a specific source name. The masks are either defined in the destination of a step, or specified when performing the step or other file movement command.

Edit masks are also used to specify refinements for step destinations, and to translate pending production secondary filenames into pending master filenames. This enables LIBRARIAN to create pending master records automatically.

There is a one–to–one correspondence between elements of a fully qualified filename. (For MPE, elements are system, file, group, and account. For UNIX, elements are system, path components, and filename.) For each element, the mask can result in carrying forward the element, replacing the element, or editing the element:

- Elements are carried forward using the equal sign (=), or the at sign (@) in a step definition, if the user can override the element.
- Elements are replaced by using a string literal.
- Elements can be edited using a combination of equal (=), at (@), question (?), minus (–) sign, and literals, as described below.

Table 3–1 describes the valid edit mask characters for any element of an MPE or UNIX filename, along with their descriptions and examples.

Table 3–1. Edit Mask Symbols and Descriptions

Edit Mask Character	Description
At sign/Star @ *	Copies original value into edited version. Typically preceded and/or followed by other characters. For example, when edited with the edit mask of ABC@XYZ, the value of FRED results in a value of ABCFREDXYZ. For MPE filenames, the result is truncated to eight characters (ABCFREDX).
Question mark ?	Copies the character at this position into the resulting string. For example, the mask ?? applied to the string FRED results in the string FR. The question mark can be combined with literal characters such as ??X, which would result in the string, FRX, or X??, resulting in XRE. It can also be combined with the minus sign (–).
Minus sign -	Indicates that the original character in that position should not be included in the edited result. For example, the mask –? applied to the string FRED results in R. Alternatively, the mask –=– results in the string RE. An additional feature is the use of "–" in conjunction with "@", which strips characters from the beginning and/or end of the original element before adding other characters to the beginning or end. For example, PRTA100 edited with – – –@S results in A100S, deleting the first three characters before adding the S:. Note that – – –=S would result in A10S, replacing the last character.
Equal sign =	Copies all remaining characters after the minus sign, question mark, and literal characters have been evaluated. For example, a mask of =X with the original string FRED results in the string FREX. The mask =Q? with the initial string FRED results in FRQD.

Edit Masks for UNIX Pathnames

To carry forward, edit, or replace an element that is at the same level in both the source and destination filenames, follow the rules described above.

Because UNIX pathnames can have varying numbers of path elements (directories), you can edit (or skip) components at varying levels in the source filename using the following construct:

/(x [– y]) [*edit–mask*]

where x and y represent the desired range of components from the source pathname. x and y are numbers from 1 to *, where * is the last directory element of the pathname. If you want a specific element, omit y which is optional.

The optional edit mask is applied to each element in the range (do not include the brackets).

For example, the mask /(1−2)/devel/!USERID/(4−*)/= applied to the filename /usr/usr2/master/screens/abc results in the filename /usr/usr2/devel/milind/screens/abc.

You can also use the following wildcards in place of x or y:

- ~ number of levels in home directory path
- . number of levels in the current working directory path
- .. one less than the number of levels in the current working directory path

You can use curly braces, i.e., { x [− y] }, to indicate mapping from the master file name rather than the current secondary file name.

For example, consider the following step called *demo–test*:

- Source files are defined as secondary files:

  sputnik:/usr/usr2/demo/dev/level1/level2/*

- Destination files are defined as secondary files:

  sputnik:/usr/usr2/demo/test/{ 5 − * } / =

  The edit mask { 5 − * } is evaluated using the associated master file path.

- Given the following source files:

  sputnik:/usr/usr2/demo/src/dir1/dir2/dir3/*

- The destination files would be expanded to the following:

  sputnik:/usr/usr2/demo/test/dir1/dir2/dir3/=

Edit Masks for Group and Accounts

You can specify an edit mask that refers to a different element (i.e., file, group or account). To do this, use the following syntax in your edit mask:

$$\left(\left\{\begin{matrix}F\\G\\A\end{matrix}\right\}start{:}length\right)$$

where F is for filename, G is for group name, and A is for account name. *Start* is the starting position, and *length* is the number of characters to be used.

The example below shows an edit mask that creates a group name using the first three characters of the filename:

Source:	ABC100S.PSOURCE
Edit:	PRG???.P(F:1:3)OBJ
Destination:	PRG100.PABCOBJ

How to Perform Steps

You can only perform steps that the LIBRARIAN Manager or Application Manager has authorized you to perform. For an online list of the steps you can perform, use the **HELP STEPS** command or open the **Steps** menu from the **File** menu.

HELP STEPS displays your user information, a list of the steps you are authorized to use, and information about the step, as shown in Figure 3–1.

STEP AUTHORIZATIONS

User ID: LIBMGR Name: Frank Phone:

Step	.Route	.Appl	Mode	Ty	From	Location			Move
AR-OUT	.AR-MAINT	.AR	R/W	MS	@	.@	.???PROD	.SYS1	COPY
AR-NEW	.AR-MAINT	.AR	R/W	SS	@	.@	.!LOGON	.SYS1	NULL
AR-IN	.AR-MAINT	.AR	R/W	SM	@	.@	.!LOGON	.SYS1	MOVE
AR-RELEASE	.AR-MAINT	.AR	R/W	SS	@	.@	.!LOGON	.SYS1	COPY
AR-COPYIN	.AR-MAINT	.AR	R/W	SM	@	.@	.!LOGON	.SYS1	MOVE

Enter HELP and the name of the Step for further information.

Figure 3–1. Step Authorizations Information

When you perform a step, LIBRARIAN authorizes the request based on the rules for that step, executes the operation for each authorized file, and logs the status of each operation in the audit database.

When LIBRARIAN authorizes a request, it authorizes each file separately, evaluating your user ID and permissions associated with it, checking the existence of the file, the policies for the file, and the rules for the step.

LIBRARIAN displays the status for requested files. The following example shows a typical "Request Status" display:

```
*REQUEST STATUS
2 authorized  0 conditional  0 violations  0 excluded
```

You have the option to review the list of files for each of the categories, and LIBRARIAN offers further online explanations of each error/exclusion status, as follows:

Authorized Files have passed all checks and can be processed as requested.

Conditional Files are conditional on read mode, and cannot be obtained in write mode because a write mode copy already exists. You have the option of obtaining the file in read mode or creating a branch. Conditional warnings are issued only when the file has serial access control.

Violations Files that did not pass one or more policy checks. For example, files cannot be outside the scope of the step. Other violations include trying to replace an existing write mode secondary, or a prestep has not been performed.

Excluded	Files have been bypassed. For example, duplicate files are excluded, as are files you excluded using the dash (–)prefix.

Note

You can suppress the "Request Status" display and associated prompts by using the **QUIET DISPLAY** command, or by selecting **Quiet Mode...DISPLAY** in the **Settings** dialog from the **User** menu.

When LIBRARIAN executes a step transaction, it performs the operation for each authorized file. At the end of the operation, LIBRARIAN displays a summary, showing the number of authorized file operations that succeeded or that failed.

Each file transaction is logged to the LIBLOG database to provide a complete audit trail. You can review the audit trail using the SHOWLOG report writer discussed in Chapter 4, "SHOWLOG Commands", in the *LIBRARIAN/iX Reference Guide*.

Figure 3–2 shows an example of a step transaction performed in LIBRARIAN (with full display – **QUIET OFF**).

```
>AP-OUT RP10S.SOURCE, RP20P.OBJECT

*AUTHORIZING *RP10S.SOURCE.LIBPROD.SYSA*
1 file(s) found
*AUTHORIZING *RP20P.OBJECT.LIBPROD.SYSA*
1 file(s) found

*REQUEST STATUS
2 authorized  0 conditional  0 violations  0 excluded

*PROCESSING REQUEST

Copied RP10S.SOURCE.LIBPROD to RP10P.LINDA.LIBDEVEL
Copied RP20P.OBJECT.LIBPROD to RP20P.LINDA.LIBDEVEL

2 file(s) copied.
0 file(s) not copied.
```

Figure 3–2. Sample LIBRARIAN Operation

Step Parameters

Each step definition includes default parameters that LIBRARIAN automatically invokes each time you perform the step. Additionally, the step definition specifies which default parameters you can override.

For example, use the **BATCH** parameter to perform the step in batch mode and schedule the step to run at a later time. Use the **MEMO** parameter to add text describing the transaction. The **BATCH** and **MEMO** options are discussed later in this chapter. Use the **NOMOVE** parameter to simulate the file operation by authorizing the request but not performing the actual operation. Use parameters to **COMPRESS** or **DECOMPRESS** destination files automatically, or to **RETAIN** copies of files when they are to be replaced.

You can review the default parameters and allowed overrides for a step by using the **HELP** command with a step name, or press **F1** with the step name highlighted on the **Steps** menu. For an offline report, run the Step Detail Report (RAD20), which describes all steps in an application. Figure 3–3 illustrates the **HELP** display for the AP-OUT step.

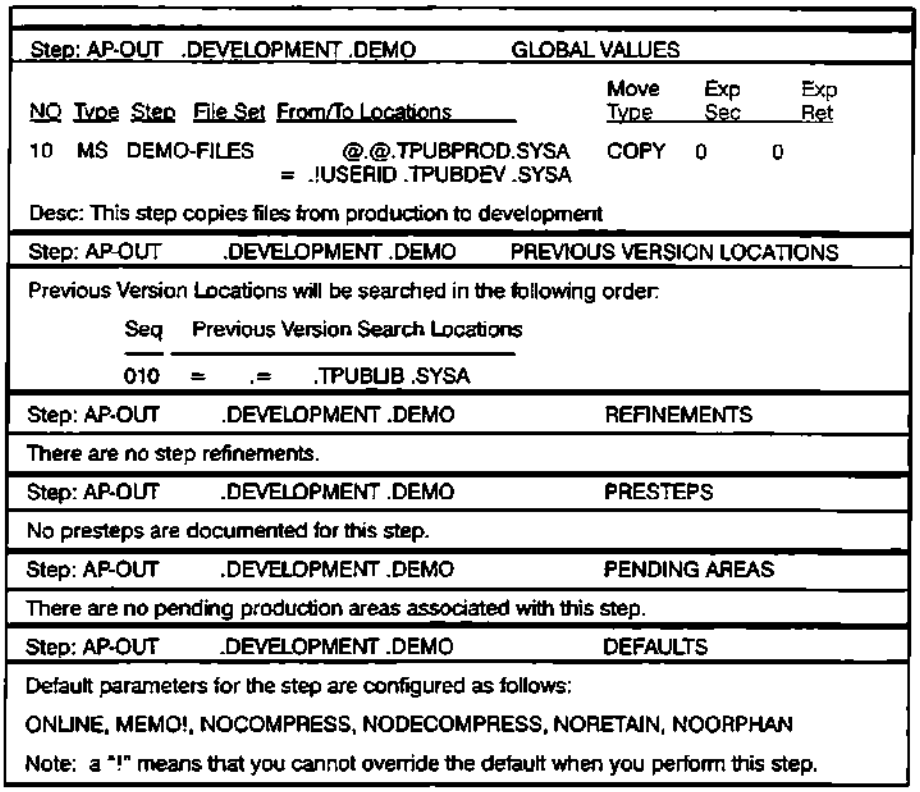


Figure 3–3. Step Information for the AP-OUT Step

Table 3–2 summarizes the parameters currently available for use with steps. For an online list of the parameters and descriptions of each, type **HELP PERFORM PARMS** at the command line, or open the **Options** menu from the step dialog and press **F1** help for a particular option.

Table 3-2. Step Parameters

Parameters		Parameters	
ANNOTATE		MODIFIED	UNMODIFIED
APPEND		NOMOVE	
AUTOUPDATE	NOAUTOUPDATE	NOSEARCH	
BATCH	NOBATCH	OLDDATE	
BRANCH		ONLINE	
COMPRESS	NOCOMPRESS	ORPHAN	NOORPHAN
CONDITIONAL = *maxcon*	NOCONDITIONAL	OWNER =	
CREATE =		PERMISSIONS =	
CREATOR = *creator*		PUSHREAD	
DECOMPRESS	NODECOMPRESS	READ	WRITE
ERRORS = *maxerr*	NOERRORS	RENUMBER	
EXTERNAL		REPLACE	
INPROGRESS		RESET	
KEEP		RETAIN	NORETAIN
LOCKWORD = *lockword*		VERIFY	
MEMO	NOMEMO	VIOLATIONS = *maxvio*	NOVIOLATIONS
MERGE =			

These parameters are described in detail under the **PERFORM** command in Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide.*

Associating Files with Projects

If projects are defined, you can associate the files processed in each step operation with a specific project. When you perform a checkout step, LIBRARIAN displays a menu of open projects for the applications that have been assigned to you. Select the appropriate project from the menu.

If you are not required to associate your work with a project you can select the "no project" option from the menu.

In addition, you can specify the project in your step command by appending the project name to the end of the step name, separated from it by a period. For example, you could perform the AP-OUT step for the REPT-MODS project:

```
>AP-OUT.REPT-MODS
```

If you are not required to associate your work with a project, you can bypass the project name and specify the "no project" option in the step command by including the special **$NP** token:

```
>AP-OUT.$NP
```

Memos

A memo is text that describes the current transaction in the audit trail. Use the **MEMO** parameter to include a memo. To create one-line memos up to 72 characters, enter the memo text on the command line as a value for the **MEMO** parameter (e.g., **MEMO** = memo-text). For multi-line memos, do not specify the text on the command line and LIBRARIAN will invoke the configured editor; the default is **EDITOR/3000** on MPE, or **vi** on UNIX. Enter at least one line of text, then exit. Enter Y when prompted to replace the memo file (MPE only).

You can review and modify the text through the SHOWLOG module.

Note

If you configured QEDIT as your editor for MPE in the configuration file, LIBRARIAN executes QEDIT enabling you to enter memo text.

Using Personal Lockwords

LIBRARIAN personal lockwords can enhance file level security in MPE while maintaining convenient file access through LIBRARIAN. You can have your own personal lockword, which is encrypted and stored in the database with your other user information. For information on defining and maintaining your own lockwords, refer to Chapter 2, "Getting Started".

If you have a personal lockword, it is placed on any file you create in LIBRARIAN. The lockword serves as protection from access by other MPE users; LIBRARIAN automatically supplies your lockword for authorized source files.

Lockwords are not automatically assigned to master files. Only Application Managers or LIBRARIAN Managers can assign lockwords to master files.

In addition to automatic lockword substitution for lockwords assigned outside of LIBRARIAN there are two ways to assign specific lockwords with LIBRARIAN:

1. Use the **SET** command to change the current lockword on files or filesets.
2. Use the **LOCKWORD** parameter to specify a lockword to use rather than your personal lockword when performing a step.

Macros

Macros are command files which can be in any location to which you have read access, or in the XEQ.OCSLIB group (MPE) or in the /opt/ocs/ocslib/xeq directory (UNIX). Use macros in place of a step to perform complex file operations. For more information on creating and using macros, refer to Chapter 9, "Macros".

Other File Operations

Additional LIBRARIAN commands are available for working with files that LIBRARIAN is tracking. Users can use these additional commands for files they own. The file owner is the LIBRARIAN user who created the file with a LIBRARIAN command. Master files are the exception — only the Application Manager or LIBRARIAN Manager can use the other file commands for master files. Some commands, such as **PRINT** and **COMPARE**, are available for master files if a user has read access.

Editing Files

It is not necessary to exit the LIBRARIAN program to work on files that you checked out. You can run any editor directly from the LIBRARIAN prompt.

Compressing Files

Compressing files can result in 60—90% disk space savings, depending on the file type. Additionally, file compression provides additional file security because compressed files cannot be read directly — they must be decompressed before they can be read (however, for MPE, LIBRARIAN offers special programs that can read compressed programs serially).

LIBRARIAN offers options for you to automatically or manually compress/decompress files.

You can use the **COMPRESS** and **DECOMPRESS** commands (also available from the **Tools** menu), or use the **COMPRESS** and **DECOMPRESS** parameters available with most file operations. Steps can also be defined to automatically compress or decompress destination files after they are created.

Certain types of files can exist that you want to exclude from compression, such as XL or program files. The LIBRARIAN Manager can use the Compress Exclusions (CE) screen to define file codes to be excluded from automatic compression.

Other Commands

Table 3–3 is a list of other LIBRARIAN commands that operate on files and describes the function of each.

Table 3-3. File Commands

Command	Function
LCOMPARE	Shows differences between text files.
COMPRESS	Compresses files.
COPY	Copies files to a new location.
DECOMPRESS	Decompresses files.
EDIT †	Accesses configured editor.
LOCK	Locks files.
MOVE	Moves files to new location.
ORPHAN	Disables tracking of secondary files.
OVERLAY	Replaces tracked files with other files.
PRINT	Displays the contents of files.
PURGE	Deletes files from the database and disk.
RELEASE †	Removes MPE security from files.
RENAME	Changes filename or fileset name.
RESET (TIMESTAMP)	Replaces modification timestamps in the database with timestamps from file labels.
RESTORE	Restores retained files to original location.
SCAN	Searches and replaces strings of text.
SCOMPARE †	Compares files with S/COMPARE.
SECURE †	Restores MPE security to files.
SET (MODE)	Changes access mode.
SET (EXPDATE)	Changes file expiration dates.
SET (LOCKWORD) †	Changes lockword on files.
SET (OWNER)	Changes the owner of a file.
UNLOCK	Releases files that were locked.
UPDATE	Refreshes read secondary from master.
VERIFY	Views information about files and versions.

† = MPE only

Operations on Untracked Files

Most LIBRARIAN file operations are restricted to files that LIBRARIAN is tracking, that is, master files or their associated secondaries and retained files. All other files on the system are untracked (unknown to LIBRARIAN).

LIBRARIAN includes a special group of **X** commands to operate on these unknown files. Table 3–4 lists the **X** commands currently available from LIBRARIAN.

Table 3-4. X Commands for Untracked Files

Command	Function
XCOMPARE †	Compares file contents using S/COMPARE
XCOMPRESS	Compresses files
XCOPY	Copies a file to a new location
XDECOMPRESS	Decompresses a file
XLCOMPARE	Shows differences between files not tracked by LIBRARIAN
XMOVE	Moves a file to a new location
XPRINT	Prints the file contents online or offline
XPURGE	Deletes a file
XRENAME	Renames a file
XSCAN	Scans/replaces text in a file
XTOUCH †	Updates the MPE modification timestamp with the current date and time

† = MPE only

When you use **X** commands, you can specify a single file, a file mask with wildcards, or a list of files or file masks, listfiles, and user filesets. The files must be untracked by LIBRARIAN.

Note

Unless you want to restrict a command to untracked files, you do not need to use the **X** prefix. LIBRARIAN will process both tracked and untracked files, but will enforce file system security for untracked files and LIBRARIAN rules for tracked files. To exclude untracked files from these operations, use the **TRACKED** parameter.

X commands allow you to specify many of the same parameters allowed for their counterpart LIBRARIAN commands.

The LIBRARIAN Manager can assign **X** capability with the User Capabilities (UC) screen. Users with **X** capability can use the **X** commands without enforcing normal file system security. Otherwise, security is enforced.

All **X** transactions are logged in the audit database, and they can be reviewed with SHOWLOG reports.

Batch Transactions

All LIBRARIAN file operations can be performed in batch mode. You can use the **BATCH** parameter on a command or step, or you can run LIBRARIAN from jobstreams.

BATCH Parameter

The **BATCH** parameter in a command or step causes the transaction to be performed in batch mode by creating a temporary job. The operation is authorized online, but executed in batch mode.

When you use the **BATCH** parameter under MPE, LIBRARIAN prompts you for a **!JOB** command and MPE:**STREAM** options. All job parameters, such as **INPRI, PRI, OUTCLASS, STREAM, AT,** and **DATE** are supported.

If the OCS-ENABLED flag is set to Y on the System Profile (SP) screen, the EXPRESS SUBMIT facility is invoked enabling you to schedule the transaction. If the flag is set to N, you are prompted to supply the login values and MPE **:STREAM** options for the transaction job before it is streamed.

When you use the **BATCH** parameter under UNIX, LIBRARIAN launches jobs using the UNIX **at** command. LIBRARIAN prompts you for **at** options, or you can set the environment variable, **LIBBATCH**, to provide these options.

Figure 3–4 illustrates the use of the **BATCH** parameter with the **COPY** command (user supplied information is shown in bold).

```
>COPY REL1.00 OF %FINPROD TO =.TEST;BATCH;OLDNAME;ALL

"AUTHORIZING "%FINPROD"
6 files found

"REQUEST STATUS
6 authorized  0 conditional  0 violation  0 excluded

"SHOW AUTHORIZED (N/Y)? N

"CONTINUE THIS OPERATION (Y/N)? Y
MPE Jobname (OCSMOVEJ): TESTJOB
User/Password (MGR):
Account/Password (PROD): PROD/TUBE
Group/Password (PUB):
Job Logon Parms (INPRI=8;PRI=DS;OUTCLASS=LP,8):
MPE STREAM Parameters (Optional); AT 03:00

If your login computer is not the host computer, where the LIBRARIAN databases are
found, please enter the System ID where the databases reside. Otherwise, just press
<RETURN>.

Host Computer System ID:

" CREATING BATCH JOBSTREAM " PROCESSING REQUEST

#J94
```

Figure 3–4. Using the BATCH Parameter

LIBRARIAN Commands in Jobstreams

You can run the LIBRARIAN program from a jobstream to perform complete transactions, including authorization, in batch mode. For example, you could set up a nightly job to move all approved files to production.

Running LIBRARIAN from a jobstream is useful when large numbers of files are involved. For example, if the fileset has 200 members and secondaries exist in several locations, the execution of a step could take more time.

By scheduling a jobstream to run late at night, you can execute transactions when the system is least busy. You can schedule a job to run after production has completed so that the files are copied to the areas where they will be used the next day.

When you run LIBRARIAN in a jobstream, remember to identify the user and password. We recommend that LIBRARIAN passwords in jobstreams be filled in dynamically when streamed through a scheduler or other stream utility that supports parameter substitution.

You can set up a job with a defined limit for errors or violations. For example, you can specify that the step can be performed only when there are no violations (**NOVIOLATIONS**), creating an "all-or-nothing" operation. You can also specify the number of times to attempt linking to a remote MPE system that is not responding with **RETRY**.

Figure 3–5 illustrates running LIBRARIAN from a jobstream to poll for approved files and move those files to the test area. The sample job SUBMITJ performs the SUBMIT step in the DEVEL route of the FINANCE application. The SUBMIT step is defined to move files from the development area to the test area if the prestep APPROVE has been performed.


```
! JOB SUBMITJ, MGR.FINTST
! CONTINUE
! LIB
USER LIBMGR:LIB
SUBMIT %FINANCE AT @.@.FINDEV; WRITE; MODIFIED
EXIT
!EOJ
```


```
#!/bin/sh
ocslib -batch<<!!!
USER LIBMGR:LIB
SUBMIT %FINANCE AT @.@.FINDEV; WRITE; MODIFIED
EXIT
!!!
```

Figure 3–5. Using LIBRARIAN Commands in a Jobstream

The sample SUBMITJ job logs on to the FINTST account and runs the LIBRARIAN program. Notice that a LIBRARIAN user and password must be supplied. Default answers are supplied automatically where the user would normally be prompted online.

LIBRARIAN then authorizes all secondaries of the step's documented fileset in the specified location. Any secondaries without write-mode access that have not been modified are not authorized. Any files with unsatisfied presteps are violations.

LIBRARIAN moves each authorized file to the specified testing location. Complete authorization and execution information appears in the job listing.

With this use of LIBRARIAN for automatic polling, LIBRARIAN authorizes any approved files and rejects files that are not ready to move as violations.

How to Check Transaction Status

LIBRARIAN sets several variables (JCWs for MPE) when executing a file operation. You can reference these values in your jobstreams or macros to control further activities. For example, assume you want a job to perform an activity that depends on the successful movement of all files in a fileset to another location. You could check the LIBFAIL variable to see if all authorized files for the previous LIBRARIAN command were moved successfully. See Table 3–5 for a list of the LIBRARIAN variables.

Table 3–5. LIBRARIAN JCWs

JCW/Variable	Function
LIBAUTHORIZED	Number of files authorized
LIBCONDITIONAL	Number of files conditional on read
LIBEXCLUDED	Number of excluded files
LIBMATCHES	Number of files in which matches were found by the **SCAN** command
LIBVIOLATIONS	Number of violations
LIBERRORS	Number of violations and conditional files
LIBFAIL	Number of files unsuccessfully processed in the last transaction
LIBOK	Number of files successfully processed in the last transaction
LIBJCW	The last LIBRARIAN error number

Reviewing File Information

The **VERIFY** command (available from the **Info** menu) offers extensive information about files. You can review information such as who last checked out a file and when, the location to which the file was copied, and what step was used to perform the copy. You can review the information online or use the **LP** parameter to send the information offline.

The **VERIFY** command accepts all file references as described for other commands. In addition, three parameters are provided for further refinement of the file selection.

OWNER = *username*	Selects only secondary files whose owner is the one you specify.
STEP = *step.route.application*	Selects files with the specified step as the most recent step performed.
MODE = *WRITE / READ*	Selects only secondaries in either read or write–mode.

When you issue the **VERIFY** command, a menu of formats displays. Each format includes different types of information. Figure 3–6 shows the **VERIFY** menu.

```
                    L I B R A R I A N   V E R I F Y   M E N U
 6 Files    0 Unknown    6 Masters    0 Secondaries    0 Retained    0 Delta
  [01]   Actual Modification Timestamp, Filecode.....  all files
  [02]   LIB Modification Timestamp, Lock Status.....  all files
  [03]   Associated Master File (or Delta File).....  all files
  [04]   Associated Master Fileset(s).....  all files
  [05]   Associated Project(s).....  all files
  [06]   Associated User Fileset(s).....  all files
  [07]   Version Information.....  all files
  [08]   Master File Counters.....  master files only
  [09]   Location of Write-Mode Copy.....  master files only
  [10]   Previous Versions (Generated Files).....  masters/secondaries
  [11]   Owner, Access Mode, Expiration, Exceptions...  secondaries only
  [12]   Last Step.....  secondaries only
  [13]   Step History.....  secondaries only
  [14]   Original File Name.....  retained files only
  [15]   Date Retained, Expiration Date.....  retained files only
  [16]   Revision Information/Tag.....  all tracked files
  [17]   Revision History.....  master files only
  [18]   Language/Description.....  master files only
  [19]   Return to LIBRARIAN prompt (or 'Q')
 Format Number [,LP]?
```

Figure 3–6. VERIFY Menu

Select a format by entering the format number and LIBRARIAN displays the requested information. You can continue to choose formats for the same files until you exit and return to the LIBRARIAN prompt.

If you are familiar with this menu, you can bypass it by using the **VERIFY** command with the **FORMAT** parameter. Figure 3–7 is a sample display of master files and associated delta files (format 3).

```
LIBRARIAN VERIFY (All Files/Associated Master File)

File                                          File Type  Master File
PENGUIN:ABC1000S.VERONICA.LIBDEVEL            S          PENGUIN:ABC1000S.SOURCE.LIBPROD
PENGUIN:ABC2000S.VERONICA.LIBDEVEL            S          PENGUIN:ABC2000S.SOURCE.LIBPROD
PENGUIN:ABC3000S.VERONICA.LIBDEVEL            S          PENGUIN:ABC3000S.SOURCE.LIBPROD
PENGUIN:ABC1000S.SOURCE.LIBPROD               M          = (DELTA FILE: D0000001  - COBOL/RPG)
PENGUIN:ABC2000S.SOURCE.LIBPROD               M          = (DELTA FILE: D0000002  - COBOL/RPG)
PENGUIN:ABC3000S.SOURCE.LIBPROD               M          = (DELTA FILE: D0000003  - COBOL/RPG)
PENGUIN:D0000001.SOURCE.LIBPROD               D          PENGUIN:ABC1000S.SOURCE.LIBPROD     OK
PENGUIN:D0000002.SOURCE.LIBPROD               D          PENGUIN:ABC2000S.SOURCE.LIBPROD     OK
PENGUIN:D0000003.SOURCE.LIBPROD               D          PENGUIN:ABC3000S.SOURCE.LIBPROD     OK
sputnik:/opt/ocs/ocslib/libdevel/             S          sputnik:/opt/ocs/ocslib/libprod/
 paul/abc1000.c                                           abc1000.c
sputnik:/opt/ocs/ocslib/libdevel/             S          sputnik:/opt/ocs/ocslib/libprod/
 paul/abc3000.c                                           abc3000.c
sputnik:/opt/ocs/ocslib/libprod/              M          =
 abc1000.c
sputnik:/opt/ocs/ocslib/libprod/              M          =
 abc3000.c
```

Figure 3–7. Sample VERIFY Display

Revisions 4

LIBRARIAN helps you track and control revisions of individual files. A *revision* is any set of changes made to a file and checked in to the library. This chapter describes how to manage revisions of files within LIBRARIAN. Topics discussed in this chapter include:

- Managing Revisions
- Identifying Revisions
- How Revisions are Stored
- Merging Revisions
- Comparing and Printing Revisions
- Purging Delta Files
- Viewing Revision Information

Managing Revisions

A *revision* refers to a file after changes have been made and checked in to the library. Each time you check out, edit, and check a file back in, LIBRARIAN assigns a new revision identifier to the file. Revision identifiers for each file reflect the number of times modifications have been made.

A revision is different from a version because it refers to a single file only. A version refers to an entire application at a specific point in time. Refer to Chapter 7, "Versions", in the *LIBRARIAN/iX Administrator's Guide*.

By default, a checkout step obtains the latest revision of a file; any previous revision is obtained by specifying the revision's identifier as a parameter to the checkout step. For example, if the current revision of MYFILE is A:4, you can obtain the A:2 revision by issuing this command:

```
>ABC-OUT MYFILE ;REV=A:2
```

Checking out an older revision of a file creates a branch. Branching is discussed later in this chapter.

Before checking out previous revisions of files, it is important to understand how revisions are named, and what happens when they are checked out.

Identifying Revisions

Revision identifiers have a version ID prefix, or an asterisk (*) prefix if no version exists. This is followed by a set of counters that uniquely identify the particular revision. Refer to Chapter 7, "Versions", in the *LIBRARIAN/iX Administrator's Guide* for more information. The counters include the version count and any number of branch/leaf count pairs delimited by periods. Revision IDs have the following general format:

```
version_id:vcount[.branch.leaf...]
```

To understand revision IDs, think of the file's revision path as a tree. The version count grows along the trunk of the tree, which starts at 0, and increments by 1 for each revision to a file within a version. If the revision ID contains only a version count, the revision is considered to be on the main development path (the trunk of the tree).

Branching occurs when you check out previous revisions of files, or force branches off the latest revision. The branch count represents the number of branches that have grown from a revision. The leaf count represents the number of revisions along a branch. Thus, the branch count and leaf count are always appended to the revision ID in pairs, allowing you to branch off the leaves of branches.

Figure 4–1 illustrates the revision history tree for a hypothetical file by showing its revision IDs.

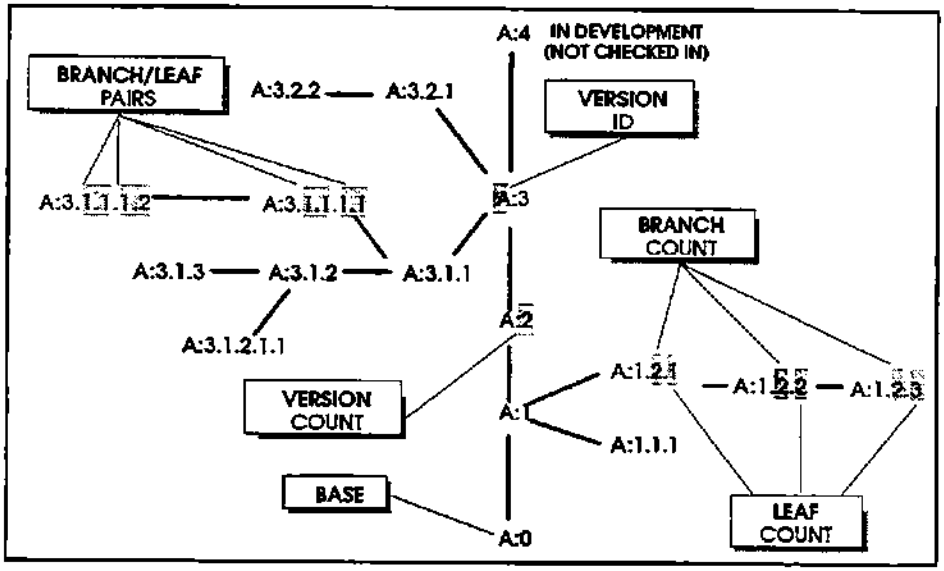


Figure 4–1. A Revision Tree

In this illustration, the version ID is A and all file identifiers start with the prefix A:. The base revision has a version count (**VCOUNT**) of 0, set when the version was defined. Subsequent revisions cause **VCOUNT** to increment by 1 with each file check in. Base revisions are protected from being flushed until you define the associated version as obsolete.

To check out a previous revision you can specify the **REVISION** parameter of the checkout step. For example, if you wish to check out revision A:1 of MYFILE, your checkout step would look something like this:

```
>ABC-OUT MYFILE ;REV=A:1
```

Branching

A *branch* is a revision that is not checked in on the main development path (trunk).

When you attempt to check out a file which has already been checked out by another user, LIBRARIAN gives you the option of accessing the file conditionally. If you answer NO, no copy will be made. However, if you respond YES to the conditional prompt, you will be prompted to indicate whether you want to create a branch.

If you respond YES, you will get a WRITE mode copy of the file on a branch. If you answer NO, you will have a READ mode copy of the file.

Branching also occurs when you checkout a previous revision of a master file, or use the **BRANCH** parameter.

In summary, branching occurs when

- a revision other than the latest trunk revision is checked out and checked in (a branch is automatically created),
- a branch is forced from the latest leaf on a branch revision, or the latest trunk revision, by specifying the **BRANCH** parameter on a checkout step.

Branching is useful to fix a problem in a previous file revision without affecting the current revision. For example, if you fix a bug in a software program and wish to send out a patch to fix a previous version of the application, you would check out the problem files (previous revisions), correct the problems, and check the files back in — automatically creating a branch. Later, you can merge these changes into the main development path as described later in this chapter.

When you create a branch, LIBRARIAN appends a branch pair to the revision ID of the revision you checked out. In Figure 4–1, if you checked out the A:1 revision, then the branch A:1.1.1 would be created. Another user checking out A:1 causes a second branch, A:1.2.1, to be created.

Checking out and checking in the most recent revision on a branch causes the leaf count to increment. The A:1.2.1 revision was checked out, revised, and checked in two more times, increasing the leaf count each time (A:1.2.2 and A:1.2.3).

Checking out previous branches of revisions causes further branches to be created. In Figure 4–2, the A:3.1.1 revision of the file was checked out, after being revised as A:3.1.2. This causes a new branch, A:3.1.1.1.1 to be created. For each branch created, a new branch count and leaf count are appended to the previous revision ID.

Forced Branching

You can force branching to happen when checking out a file. This can be useful when you know someone else will need the file in write–mode, and you do not want your changes to be reflected on the main development path. For example, if you want to check out the current revision of the file MYFILE and force a branch, issue the following command:

```
>ABC-OUT MYFILE ;BRANCH
```

Assuming that MYFILE has the revision tree illustrated in Figure 4–2, the most current revision (in this case, A:3) is checked out as a new branch. In this example, A:3.1.1 is created.

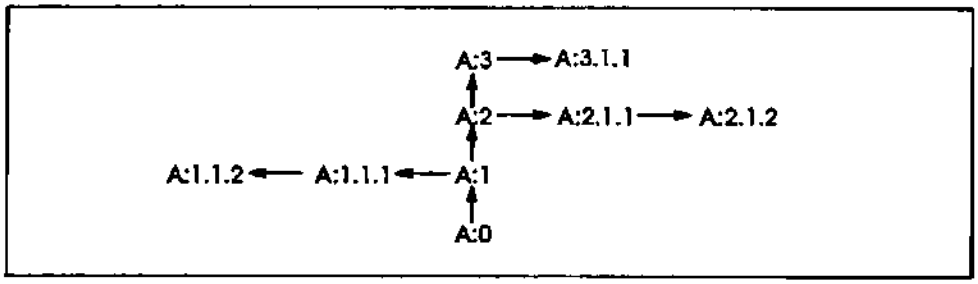


Figure 4–2. Revision Tree for MYFILE

If any write–mode copies exist, the only method of obtaining another write–mode copy is to create a branch. Branches can later be merged into the main development path as described in "Revisions" later in this chapter.

Preventing Branching

You can use the **NOBRANCH** parameter to block the use of the **BRANCH** option. The LIBRARIAN or the Application Manager can prevent users from using **BRANCH** by coding **NOBRANCH** in a macro, or by configuring this option as an additional step parameter on the STO screen.

The **NOBRANCH** option prevents the branch prompt from appearing and prevents a user from using the **BRANCH** parameter. Additionally, **NOBRANCH** prevents the user from checking out a write–mode copy of a previous revision of a file.

New Files

New files are files that are introduced "new" to LIBRARIAN in a development or test location by a step with a Pending Production Area (see PP screen in the *LIBRARIAN Reference Guide*), rather than being checked out from the library. These files may be entirely new programs or other files that do not exist in the library, or may simply be new generations of files that were not checked out but will be checked in "on top of" existing master files.

New files that will not replace existing master files are automatically assigned a revision number of

<current version>:1

by LIBRARIAN. The idea here is that these files are part of the current version, but are not "base revisions" (*<current version>*:0), since base revisions by definition were there when the version was created.

New files that will replace an existing master file are assigned the next available revision number on the trunk – i.e.,

<current version>:*<nextvcount>*

as though the current master file had been checked out.

New files, then, cannot be branched in either case. If you branch a source file, and want the related executable file to bear the same revision number as the source, you must check out the executable file as well as the source, branching to create the desired revision number.

How Revisions Are Stored

Previous revisions of files must be retained if they are going to be recovered or modified later. The base revision of a file for a version is automatically retained. Intermediate revisions are retained if the step **RETAIN** parameter is in effect. Refer to Chapter 4, "File Movement Rules", in the *LIBRARIAN/iX Administrator's Guide* for more information about setting the **RETAIN** parameter on step definitions.

Revisions are typically retained in the master library. You can also retain files in secondary locations by using the **RETAIN** parameter on a step that creates files in a secondary location where the file already exists.

In addition to a file's revision identifier, a generation count is recorded. The generation count (**GCOUNT**) begins at 1 when the file is created, and increments by 1 each time the file is replaced by a new revision. All revisions on the same branch off of the trunk have the same **GCOUNT**. You can only identify branches using revision IDs.

Delta Files vs. Generation Files

Revisions to text files can be retained either in delta files or as generation files.

- A *delta file* is a special file containing the complete text of the first source file revision and a history of all subsequent changes — i.e., insertions and deletions, who made the changes, and when they were made. Only revisions to text files can be retained as deltas; all other files are retained as generation files.
- A *generation file* is a complete compressed archive copy of an older revision of a file.

By default, retained files are stored as generation files. If Delta is set to Y for an application on the Applications (AP) screen, previous revisions of text files within that application are stored as deltas.

Most users do not need to know whether revisions are being stored as generation files or delta files. Delta files take less space than the corresponding generation files and enable the use of LIBRARIAN's merge and annotation feature.

LIBRARIAN tracks revisions using a system generated name in the format G####### (MPE) or .g######## (UNIX). These unique G-names are derived randomly. The G-name for a revision appears on reports and on LIBRARIAN screens. For CM/KSAM files (MPE), the key file, if not compressed, is stored as C#######.

Delta file names are in the format D####### (MPE) or .d######## (UNIX). Delta file numbering is sequential.

Location of Retained Files

Generation files typically reside in the same location as the corresponding master file; you can move these files to another location by using the **MOVE** command. You can only move delta files if you also move the corresponding master file.

Managing Generation and Delta Files

You can compress generation files to conserve disk space. If you want to have LIBRARIAN compress these files automatically, use the System Profile (SP) screen to set Auto-Compress Retained Files to "Y".

When you want to distribute these compressed generation files using the **REVISION** and **TAG** parameters, be sure to use the **DECOMPRESS** parameter. This ensures that these files do not remain compressed in the production location.

You can use the FLUSH utility and **PURGE** command to delete revisions that you no longer need. Keep in mind that the base revision of a file is kept until you make the version to which it belongs obsolete. When you flush obsolete revisions from delta files, the delta file usually becomes smaller.

All other retained files have expiration dates, and you can flush them when they have expired. The expiration date for files created with steps is determined by the number of days you specified on the Steps (ST) screen. When you use the **RETAIN** parameter in a command, the files created by the command expire immediately. You can change the expiration date for a file by using **SET EXPDATE**. If you want to purge expired files, run the FLUSH utility, as described in Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide.*

In addition to the expiration date of the retained file, you can specify the minimum number of generation files for a master file that should be kept by using the System Profile screen (SP). The FLUSH POLICY field on the SP screen allows you to enter a number from 0 to 99 for the number of previous generations that should be retained. If a retained revision has expired, but is within the flush policy limit, the retained file is kept.

Warning

The FLUSH policy applies to generations along the trunk. If a trunk revision qualifies to be flushed, then all revisions on a branch from that file are also flushed.

Merging Revisions

To merge revisions, use the **MERGE** parameter. Merge is only available when revisions are stored as deltas. Merge is also restricted to master–to–secondary (checkout) steps. This allows you to resolve any conflicts and test the result prior to introducing the merged file into the library.

Note

You can only merge revisions of the same file. You cannot merge two different files (i.e., files must come from the same base).

For example, suppose Figure 4–2 shows the current revision tree for MYFILE.

If you want to merge the changes from A:1.1.2 and A:2.1.2 with A:3 to create A:4, check out A:3 as follows:

```
>ABC-OUT MYFILE ;MERGE=A:1.1.2, A:2.1.2
```

Figure 4–3 illustrates merging two branch revisions into the latest trunk revision.

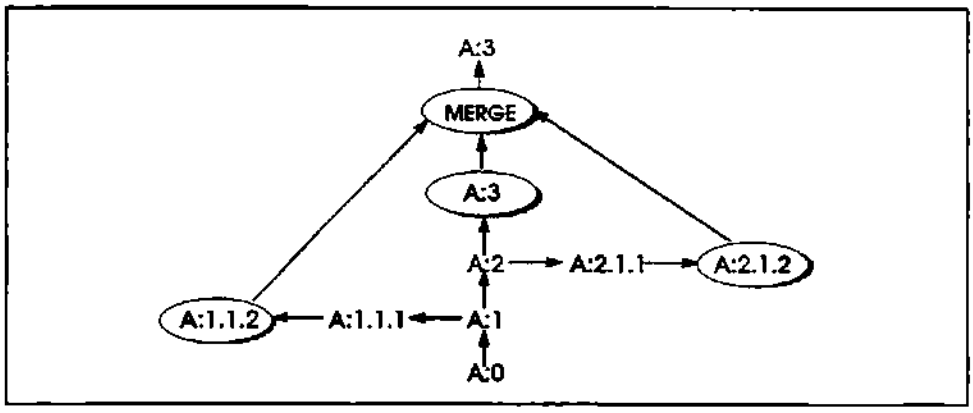


Figure 4–3. Merging Two Branches into the Trunk

Merging Specific Revisions

You can merge changes from a specific revision by using the exclamation point (!) in your merge list. This includes only changes made in that revision, ignoring previous changes along that branch. For example:

```
>ABC-OUT MYFILE ;MERGE=!A:1.1.2, A:2.1.2
```

Figure 4–4 illustrates merging a specific set of changes. The solid arrows indicate changes included in this merge. Changes made between revisions A:1 and A:1.1.1 are not included in the merge.

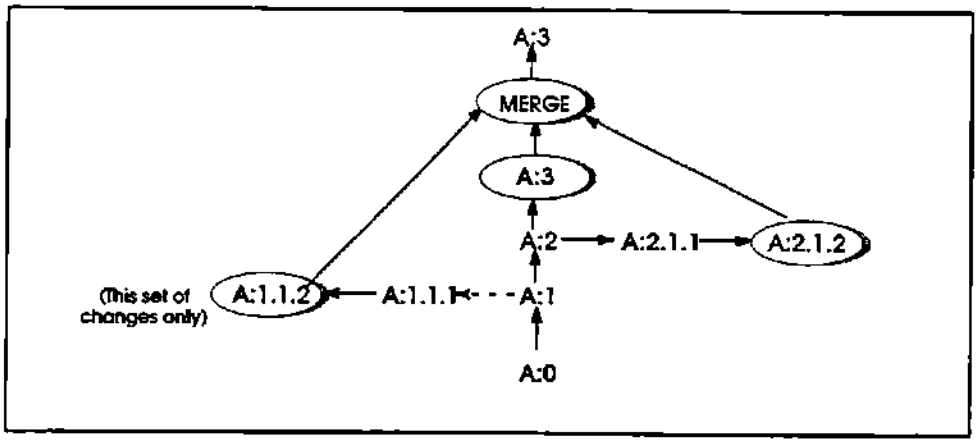


Figure 4-4. Merging a Specific Revision

Excluding Revisions from a Merge

You can exclude specific changes when merging revisions using the minus (–) sign in your merge list. This includes all changes along the development path, except for the specified revision. For example:

```
>ABC-OUT MYFILE ;MERGE=A:1.1.2, A:2.1.2, -A:2.1.1
```

Figure 4–5 illustrates excluding a set of changes from a merge. The solid arrows indicate changes included in this merge.

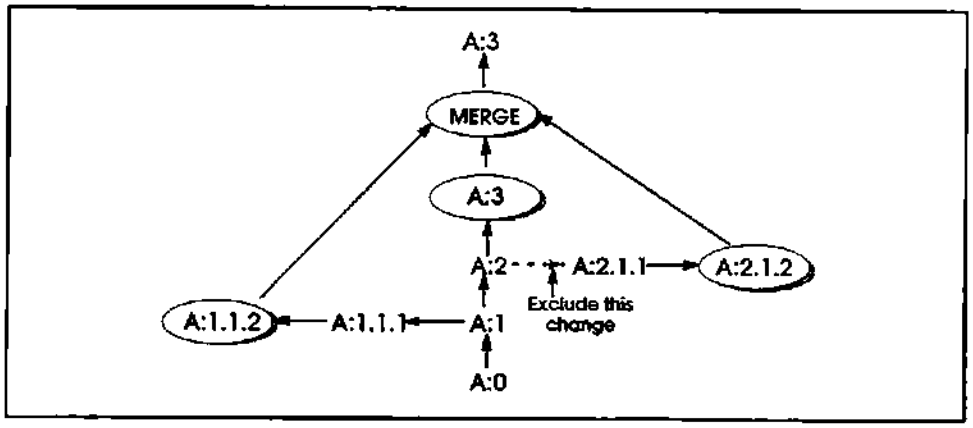


Figure 4–5. Merging Two Branches with Exclusions

Resolving Conflicts

LIBRARIAN notifies you if it encounters a conflict during a merge. If LIBRARIAN informs you that it encountered changes affecting the same part of the code (e.g., you changed a line in one revision that was deleted in another revision being merged), you must decide whether to retain the insertion or the deletion.

LIBRARIAN annotates the conflicting blocks with comments in the format for the language of that file. For more details on language, refer to the Fileset (FS) screen in Chapter 5, "Screens", in the *LIBRARIAN/iX Reference Guide*.

Using your editor, search the merged file (in the development area) for the string <=?=> to locate conflicts. Figure 4–6 contains sample conflict notation for a COBOL file.

```
001000                  IDENTIFICATION DIVISION.
001100                  PROGRAM-ID. CALLREAD.
001200                  ENVIRONMENT DIVISION.
001300                  INPUT-OUTPUT SECTION.
001400                  FILE-CONTROL.
001500  *<=?=>          ??????  Merge Conflict ??????  <=?>
001600  *<=I=>          Revision A:1 [08/23/91 14:37 DEREK]
001700  *                       SELECT OUTPUT-FILE ASSIGN TO "OUTPUT".
001800  *<=E=>          Revision A:1 [End INSERT]
001900  *<=?=>          ?????? End Merge Conflict ??????  <=?=>
002000                         SELECT OUTPUT-FILE ASSIGN TO "LISTING".
002100                  DATA DIVISION.
002200                  FILE SELECTION.
002300                  FD OUTPUT-FILE
002400                       LABEL RECORDS ARE OMITTED.
002500                  01 RECORD-BUFFER                  PIC X(79).
002600                  WORKING-STORAGE SECTION.
002700                  01 FILE-NAME                      PIC X(38) VALUE SPACE
002800                  01 CALL-STATUS                    PIC S9(4) COMP VALUE 0
002900                  01 RECEIVE-BUFFER-AREA:
003000                       05 RECEIVE-BUFFER            PIC X(79).
003100                       05 FILLER                    PIC X(4019).
003200                  PROCEDURE DIVISION.
003300                  PARAGRAPH-1.
003400
003500                       OPEN OUTPUT OUTPUT-FILE.
003600                       DISPLAY "ENTER FILE TO READ:".
003700                       ACCEPT FILE-NAME.
003800                       DISPLAY "READING " FILE-NAME.
003900
004000                       PERFORM PARAGRAPH-2
004100                            UNTIL CALL-STATUS NOT EQUAL ZERO.
004200
004300                       DISPLAY " ** ENDING DEMOREAD".
004400                       STOP RUN.
004500
004600                  PARAGRAPH-2.
004700
004800                       CALL "SUB100" USING @FILE-NAME, @RECEIVE-BUFFER,
004900                                           CALL-STATUS.
005000
005100                       IF CALL-STATUS EQUAL ZERO
005200                            WRITE RECORD-BUFFER FROM RECEIVE-BUFFER.
```

Figure 4–6. Sample Conflict Notation

Comparing and Printing Revisions

You can compare revisions using the **FROMREV** and **TOREV** parameters with the **LCOMPARE** and **SCOMPARE** commands. For example, to compare the second and base revisions of MYFILE, type:

```
>LCOMPARE MYFILE ;FROMREV=A:2 ;TOREV=A:0
```

Figure 4–7 shows the **LCOMPARE** report highlighting changes between revisions A:0 and A:2. This report can be generated online or printed to an offline device.

```
                        LIBRARIAN File Difference Listing

Reference File     : ABC1000S.MASTER.LIB400 [A:0]   MON, AUG 19,1991, 11:16 AM
Compare File       : ABC1000S.MASTER.LIB400 [A:2]                      Page 1

          001000    IDENTIFICATION DIVISION.
DELETE    002000    PROGRAM-ID. CALLREAD.
INSERT    002000    PROGRAM-ID. DEMOREAD.
          003000    ENVIRONMENT DIVISION.
          004000    INPUT-OUTPUT SECTION.
          005000    FILE-CONTROL.
          006000        SELECT OUTPUT-FILE ASSIGN TO "OUTPUT".
INSERT    007000        SELECT INPUT-FILE ASSIGN TO "INPUT".
          008000    DATA DIVISION.
          009000    FILE SECTION.
          010000    FD OUTPUT-FILE
          011000        LABEL RECORDS ARE OMITTED.
          012000    01 RECORD-BUFFER   PIC X(132).
INSERT    013000    FD INPUT-FILE
INSERT    014000        LABEL RECORDS ARE OMITTED.
INSERT    015000    01 RECORD-BUFFER  PIC X(79).
          016000    WORKING-STORAGE SECTION.
          017000    01 FILE-NAME                        PIC X(38) VALUE SPACE.
          018000    01 CALL-STATUS                      PIC S9(4) COMP VALUE 0
          019000    01 RECEIVE-BUFFER-AREA:
          020000        05 RECEIVE-BUFFER               PIC X(79).
          021000        05 FILLER                       PIC X(4019).
          022000    PROCEDURE DIVISION.
          023000    PARAGRAPH-1.
          024000
*****************************************
** 0008 MATCHING LINES NOT DISPLAYED **
*****************************************
          033000        DISPLAY " **ENDING DEMOREAD".
          034000        STOP RUN.
          035000
          036000    PARAGRAPH-2.
          037000
          038000        CALL "SUB100" USING @FILE-NAME, @RECEIVE-BUFFER,
          039000                            CALL-STATUS.
          040000
          041000        IF CALL-STATUS EQUAL ZERO
          042000          WRITE RECORD-BUFFER FROM RECEIVE-BUFFER.
```

Figure 4-7. LCOMPARE Offline Printout

Annotated Listings

You can create a listing that highlights example of the changes that were made for each revision of a file using the **ANNOTATE** parameter with the **PRINT** command if revisions are stored in a delta file. For example, the following command produces the printout shown in Figure 4–8.

```
>PRINT ABC1000S.MASTER ;REV=A:2 ;ANNOTATE
```

For information on command syntax and usage, refer to Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide*.

For information on printing, comparing, and scanning files, refer to Chapter 5, "Printing, Scanning, and Comparing Files".

```
FILENAME:  ABC1000S.MASTER.LIB400 [A:2]

001000     IDENTIFICATION DIVISION.
002000     PROGRAM-ID. CALLREAD.                          <--DELETE Rev A:1 [08/16/91 12:37:38 DEREK]
003000     PROGRAM-ID. DEMOREAD.                          <--INSERT Rev A:1 [08/16/91 12:37:38 DEREK]
004000     ENVIRONMENT DIVISION.
005000     INPUT-OUTPUT SECTION.
006000     FILE-CONTROL.
007000               SELECT OUTPUT-FILE ASSIGN TO "OUTPUT".
008000               SELECT INPUT-FILE ASSIGN TO "INPUT".     <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
009000     DATA DIVISION.
010000     FILE SECTION.
011000     FD OUTPUT-FILE
012000               LABEL RECORDS ARE OMITTED.
013000     01 RECORD-BUFFER              PIC X(132).
014000     FD INPUT-FILE                                  <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
015000               LABEL RECORDS ARE OMITTED.           <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
016000     01 RECORD-BUFFER              PIC X(79).       <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
017000     WORKING-STORAGE SECTION.
018000     01 FILE-NAME                  PIC X(38) VALUE SPACE.
019000     01 CALL-STATUS                PIC S9(4) COMP VALUE 0
020000     01 RECEIVE-BUFFER-AREA:
021000               05 RECEIVE-BUFFER          PIC X(79).
022000               05 FILLER                  PIC X(4019).
023000     PROCEDURE DIVISION.
024000     PARAGRAPH-1.
025000
026000               OPEN OUTPUT OUTPUT-FILE
027000               DISPLAY "ENTER FILE TO READ:".
028000               ACCEPT FILE-NAME.
029000               DISPLAY "READING " FILE-NAME.
030000
031000               PERFORM PARAGRAPH-2
032000                         UNTIL CALL-STATUS NOT EQUAL ZERO.
033000
034000               DISPLAY " ** ENDING DEMOREAD"
035000               STOP RUN.
036000
037000     PARAGRAPH-2.
038000
039000               CALL "SUB100" USING @FILE-NAME, @RECEIVE-BUFFER,
040000                                   CALL STATUS.
041000
042000               IF CALL-STATUS EQUAL ZERO
043000                         WRITE RECORD-BUFFER FROM RECEIVE-BUFFER.
```

Figure 4–8. PRINT with ANNOTATE Parameter

Purging Delta Files

If you purge a master file that has an associated delta file, the delta file is not automatically purged. This enables restoration of the master file at a later time. To purge the delta file, use the **DELTA** parameter with the **PURGE** command. For example, purge MYFILE and its corresponding delta file by typing:

```
>PURGE MYFILE ;DELTA
```

This command deletes the delta file and the master file.

Viewing Revision Information

You can review revision information by using the **VERIFY** command. For example, to view information for files in the @.MASTER.LIB400 location, type:

```
>VERIFY @.MASTER.LIB400
```

This command produces the menu shown in Figure 4–9.

```
                    L I B R A R I A N   V E R I F Y   M E N U
------------------------------------------------------------------------------
  6 Files    0 Unknown    6 Masters    0 Secondaries    0 Retained    0 Delta
==============================================================================
   [01]   Actual Modification Timestamp, Filecode.....  all files
   [02]   LIB Modification Timestamp, Lock Status.....  all files
   [03]   Associated Master File (or Delta File).....  all files
   [04]   Associated Master Fileset(s).....  all files
   [05]   Associated Project(s).....  all files
   [06]   Associated User Fileset(s).....  all files
   [07]   Version Information.....  all files
   [08]   Master File Counters.....  master files only
   [09]   Location of Write-Mode Copy.....  master files only
   [10]   Previous Versions (Generated Files).....  masters/secondaries
   [11]   Owner, Access Mode, Expiration, Exceptions...  secondaries only
   [12]   Last Step.....  secondaries only
   [13]   Step History.....  secondaries only
   [14]   Original File Name.....  retained files only
   [15]   Date Retained, Expiration Date.....  retained files only
   [16]   Revision Information/Tag.....  all tracked files
   [17]   Revision History.....  master files only
   [18]   Language/Description.....  master files only
   [19]   Return to LIBRARIAN prompt (or 'Q')
  Format Number [,LP]?
```

Figure 4–9. VERIFY Menu

Formats 16 and 17 show current revision/tag information and revision history. A related display is format 3, which lists the files and delta file information for a set of master files.

Figure 4–10 shows format 16, revision information, for the ABC application and Figure 4–11 shows format 17 for these five files.

```
LIBRARIAN VERIFY (Masters-Secondaries/Revision Information)

File                                        Latest Revision/Tag
------------------------------------------  -------------------
PENGUIN:ABC1000S.SOURCE.LIBPROD             V.2.00:1
                                            PATCH-201
PENGUIN:ABC2000S.SOURCE.LIBPROD             V.2.00:2
PENGUIN:ABC3000S.SOURCE.LIBPROD             V.2.00:1
                                            PATCH-201
sputnik:/opt/ocs/ocslib/libprod/            V.2.00:2
 abc1000.c
sputnik:/opt/ocs/ocslib/libprod/            V.2.00:3
 abc2000.c                                  PATCH-UK-201
sputnik:/opt/ocs/ocslib/libprod/            V.2.00:2
 abc3000.c
```

Figure 4–10. Master-Secondary Revision Data (VERIFY Format 16)

```
LIBRARIAN VERIFY (Master Files/Revision History)

Master File
  Revision                          Project     Tag         Date/Time
PENUGIN:ABC2000S.SOURCE.LIBPROD
  V.2.00:2                          SR2935      PATCH200    DEC 15, 1993,  5:05 PM
  V.2.00:1.1.1                      SR9210                  DEC 13, 1993,  2:40 PM
  V.2.00:1                          SR9210      PATCH101    DEC 12, 1993, 10:06 AM
  V.2.00:0                                                  DEC 11, 1993,  1:22 PM
  *:1                                                       NOV  8, 1993,  4:55 PM
sputnik:/opt/ocs/ocslib/libprod/abc2000.c
  V.2.00:3                          SR9835                  MAR 10, 1994,  1:07 PM
  V.2.00:2.2.1                      SR9670                  PENDING
  V.2.00:2.1.2                      SR9622                  MAR  5, 1994,  2:42 PM
  V.2.00:2.1.1                      SR9510      PATCH200    FEB 14, 1994, 11:07 AM
  V.2.00:2                          SR9211                  FEB  6, 1994, 10:34 AM
  V.2.00:1                          SR9210      PATCH101    FEB  2, 1994,  9:18 AM
  V.2.00:0                                                  JAN 22, 1994,  2:45 PM
```

Figure 4-11. Revision History (VERIFY Format 17)

Figure 4–12 shows the delta files associated with the ABC application using format 3.

```
LIBRARIAN VERIFY (All Files/Associated Master File)

File                                  File Type  Master File
PENGUIN:ABC1000S.VERONICA.LIBDEVEL    S   PENGUIN:ABC1000S.SOURCE.LIBPROD
PENGUIN:ABC2000S.VERONICA.LIBDEVEL    S   PENGUIN:ABC2000S.SOURCE.LIBPROD
PENGUIN:ABC3000S.VERONICA.LIBDEVEL    S   PENGUIN:ABC3000S.SOURCE.LIBPROD
PENGUIN:ABC1000S.SOURCE.LIBPROD       M   * (DELTA FILE: D0000001  - COBOL/RPG)
PENGUIN:ABC2000S.SOURCE.LIBPROD       M   * (DELTA FILE: D0000002  - COBOL/RPG)
PENGUIN:ABC3000S.SOURCE.LIBPROD       M   * (DELTA FILE: D0000003  - COBOL/RPG)
PENGUIN:D0000001.SOURCE.LIBPROD       D   PENGUIN:ABC1000S.SOURCE.LIBPROD     OK
PENGUIN:D0000002.SOURCE.LIBPROD       D   PENGUIN:ABC2000S.SOURCE.LIBPROD     OK
PENGUIN:D0000003.SOURCE.LIBPROD       D   PENGUIN:ABC3000S.SOURCE.LIBPROD     OK
sputnik:/opt/ocs/ocslib/libdevel/     S   sputnik:/opt/ocs/ocslib/libprod/
 paul/abc1000.c                           abc1000.c
sputnik:/opt/ocs/ocslib/libdevel/     S   sputnik:/opt/ocs/ocslib/libprod/
 paul/abc3000.c                           abc3000.c
sputnik:/opt/ocs/ocslib/libprod/      M   =
 abc1000.c
sputnik:/opt/ocs/ocslib/libprod/      M   =
 abc3000.c
```

Figure 4–12. Version Data (VERIFY Format 3)

On format 3, note the checksum status on the far right. If the status is OK, LIBRARIAN calculated the checksum and found your delta file's integrity is okay. If you see ** or ER in the checksum column, call OCS Customer Service. This error indicates the checksum is incorrect or LIBRARIAN cannot calculate the checksum, and you may have an integrity problem.

Revision Reports

Revision information is available in standard LIBRARIAN reports. Table 4–1 lists the reports you can use to access revision data.

Table 4–1. Revision Information in Standard Reports

Report Code	Title	Description
RRH10	Revision History	Revision history for files.
RVD10	File Version Report	Detailed information on all files in a version.
RVT10	Version Timestamp Report	Version and timestamp information for each file in an application.
RVT20	Version Timestamp Exceptions	Version and timestamp information for all files in an application that have changed outside of LIBRARIAN control.

Printing, Scanning, and Comparing Files 5

In addition to powerful file movement capabilities, LIBRARIAN offers tools that allow you to scan and compare the contents of files.

This chapter describes how to print and display files, scan and replace strings of text, and view the differences between files.

Operations discussed in this chapter include:

- Printing Files
- Scanning and Replacing Text
- Comparing Files with LCOMPARE
- Comparing files with SCOMPARE

Printing Files files

You can view the contents of files directly from LIBRARIAN with the **PRINT** command (also available from the **Tools** menu), which displays the files at your terminal or offline. For example, the following command displays the contents of a file at the terminal with lines numbered:


```
>PRINT SYSA:ABC.PUB.FIN;NUMBERED

>PRINT /usr/finldevel/data/abc;NUMBERED
```

The contents of the file are displayed one screen at a time. At the end of each screen, you can respond to the prompt to continue or quit. The prompt includes the line number of the next line to be displayed and the total number of lines in the file. You can proceed directly to any line by specifying a line number at the prompt, or you can exit by typing the letter N. For example:

```
>Continue (23/4825)? 367
```

will take you directly to line 367. You can only view files to which you have read access.

Note 

You can print QEDIT files (FILECODE = 111) if you are using QEDIT Version 4.L.55 or higher.

Annotation

If you are using delta files to store revisions, you can produce an annotated printout of your files that shows deletions, insertions, and revision information such as time and date of the change. For example, use the **ANNOTATE** parameter with the **PRINT** command to view all changes made through the second revision of the file A3.MASTER by typing:


```
>PRINT A3.MASTER.LIB400 ;REV=A:2 ;ANNOTATE ;OFFLINE

>PRINT /usr/master/lib400/a3;REV=A:2;ANNOTATE;OFFLINE
```

This produces the printout shown in Figure 5–1.

```
FILENAME: ABC1000S.MASTER.LIB400 [A:2]

001000      IDENTIFICATION DIVISION.
002000      PROGRAM-ID.CALLREAD.                          <--DELETE Rev A:1 [08/16/91 12:37:38 DEREK]
003000      PROGRAM-ID. DEMOREAD.                         <--INSERT Rev A:1 [08/16/91 12:37:38 DEREK]
004000      ENVIRONMENT DIVISION.
005000      INPUT-OUTPUT SECTION.
006000      FILE-CONTROL.
007000            SELECT OUTPUT-FILE ASSIGN TO "OUTPUT".
008000            SELECT INPUT-FILE ASSIGN TO "INPUT".    <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
009000      DATA DIVISION.
010000      FILE SECTION.
011000      FD OUTPUT-FILE
012000            LABEL RECORDS ARE OMITTED.
013000      01 RECORD-BUFFER          PIC X(132).
014000      FD INPUT-FILE                                 <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
015000            LABEL RECORDS ARE OMITTED.              <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
016000      01 RECORD-BUFFER          PIC X(79).         <--INSERT Rev A:2 [08/16/91 12:37:28 DEREK]
017000      WORKING-STORAGE SECTION.
018000      01 FILE-NAME              PIC X(36) VALUE SPACE.
019000      01 CALL-STATUS            PIC S9(4) COMP VALUE 0
020000      01 RECEIVE-BUFFER-AREA.
021000            05 RECEIVE-BUFFER        PIC X(79).
022000            05 FILLER                PIC X(4019).
023000      PROCEDURE DIVISION.
024000      PARAGRAPH-1.
025000
026000            OPEN OUTPUT OUTPUT-FILE
027000            DISPLAY "ENTER FILE TO READ:".
028000            ACCEPT FILE-NAME.
029000            DISPLAY "READING " FILE-NAME.
030000
031000            PERFORM PARAGRAPH-2
032000                  UNTIL CALL-STATUS NOT EQUAL ZERO.
033000
034000            DISPLAY " " ENDING DEMOREAD"
035000            STOP RUN.
036000
037000      PARAGRAPH-2.
038000
039000            CALL "SUB100" USING @FILE-NAME, @RECEIVE-BUFFER,
040000                              CALL STATUS.
041000
042000            IF CALL-STATUS EQUAL ZERO
043000                  WRITE RECORD-BUFFER FROM RECEIVE-BUFFER.
```

Figure 5–1. PRINT Offline Printout

Scanning and Replacing Text

The **SCAN** command (also available from the **Tools** menu) searches text, binary, and compressed files for character strings; it optionally replaces those strings of text. This command is a powerful tool for reviewing file contents online, scanning files for text strings, and/or incorporating global changes across large groups of files. You can search the entire file or only search specific line or column ranges.

You can scan files if you have read access, and can replace text only if you have write access. This restriction does not apply to Librarian Managers and Application Managers for files in their applications. In addition, users with the LIBRARIAN X capability do not need these permissions for files that are not being tracked by LIBRARIAN.

Note 

You can print QEDIT files (FILECODE = 111) if you are using QEDIT Version 4.L.55 or higher.

You can search for a specific string of characters, or use special wildcards for pattern matching. Additionally, you can include an associated replacement string and invoke a prompt to confirm each replacement.

Note 

Enclose the search string (*search*) in quotes only if it includes commas, semicolons, slashes, or blanks.

The following pattern-matching wildcards can appear anywhere in the search string:

@	match any number of any character
?	match any single alphanumeric character
#	match any single numeric character
*	match any single alphabetic character
^	match any single blank character
!	match any single character
{...}	match a character in the set of characters enclosed in braces (e.g., {ABC}). You can reference a maximum of ten character sets in a single command.

All pattern-matching wildcards (except for @) can be followed by +, indicating a match for one or more occurrences. A minus sign (-) following the wildcard indicates zero or more occurrences. For example, the search string #+ informs LIBRARIAN to search for a string containing one or more consecutive numeric characters.

The following characters can be used at the beginning and end of search strings, respectively:

[	match string at beginning of line only.
]	match string at end of line only.

The backslash (\) can precede any pattern-matching character and itself to indicate a literal match.

Examples

The following command searches all source files in the finance application for all occurrences of the string $INCLUDE, without sensitivity to case, and lists all of the lines where a match is found.

```
>SCAN @.SOURCE.FIN; TEXT=$INCLUDE; IGNORE
>SCAN /usr/fin/source/*.pas; TEXT=$INCLUDE;IGNORE
```

The following example searches the FIN fileset for all files that include references to version 2 (e.g., VER 2.00, 2.01, 2.02, etc.). The metacharacters # in the command indicate any numeric value. The command specifies one match so that the scanning of each file stops after locating one reference to version 2.

The **LISTFILE** parameter directs the program to create a listfile that includes the names of all files where a match was found. Then, you can specify the listfile name in a LIBRARIAN command to move or copy all files containing references to version 2.

```
>SCAN %FIN; TEXT="VER = '2.##'"; MATCHES=1;LISTFILE=V2LST.PUB
>SCAN %FIN; TEXT="VER = '2.##'", MATCHES=1;LISTFILE=v2lst
```

Replacement Variables

You can use variables instead of literal text as the replace string(s). These variables include:

!GCOUNT	Substitutes the generation count for this file.
!NEXTG	Substitutes the next generation count for this file.
!NEXTV	Substitutes the next version count for this file.
!REVISION	Substitutes the revision ID for this file.
!VCOUNT	Substitutes the version count for this file.
!VERSION	Substitutes the version name for this file.

Note

You cannot replace text in master files that have revisions stored as deltas.

Edit masks can also be used to control replacement as described at the beginning of Chapter 1, "Commands" in the *LIBRARIAN/iX Reference Guide*. Enclose the edit mask in parentheses.

If you use a prefix of "+" (plus) with a replace string, LIBRARIAN will append the string to the line on which a match was found, rather than replace the matching string.

You can use a special replacement variable, **!DELETE**, instead of a replacement string to physically delete lines from a file that contain a match.

Comparing Files with LCOMPARE

Use the **LCOMPARE** command (also available from the **Tools** menu) to see the differences between files. You can compare a development copy to its master, a master to a previous revision, or any unrelated files. In addition, you can compare compressed files.

You can use **LCOMPARE** to compare physical files, logical filesets, and groups of physical files. You have access to all of the file specification options that LIBRARIAN offers for other commands, and you have both an enhanced 80-column online display and a standard offline report.

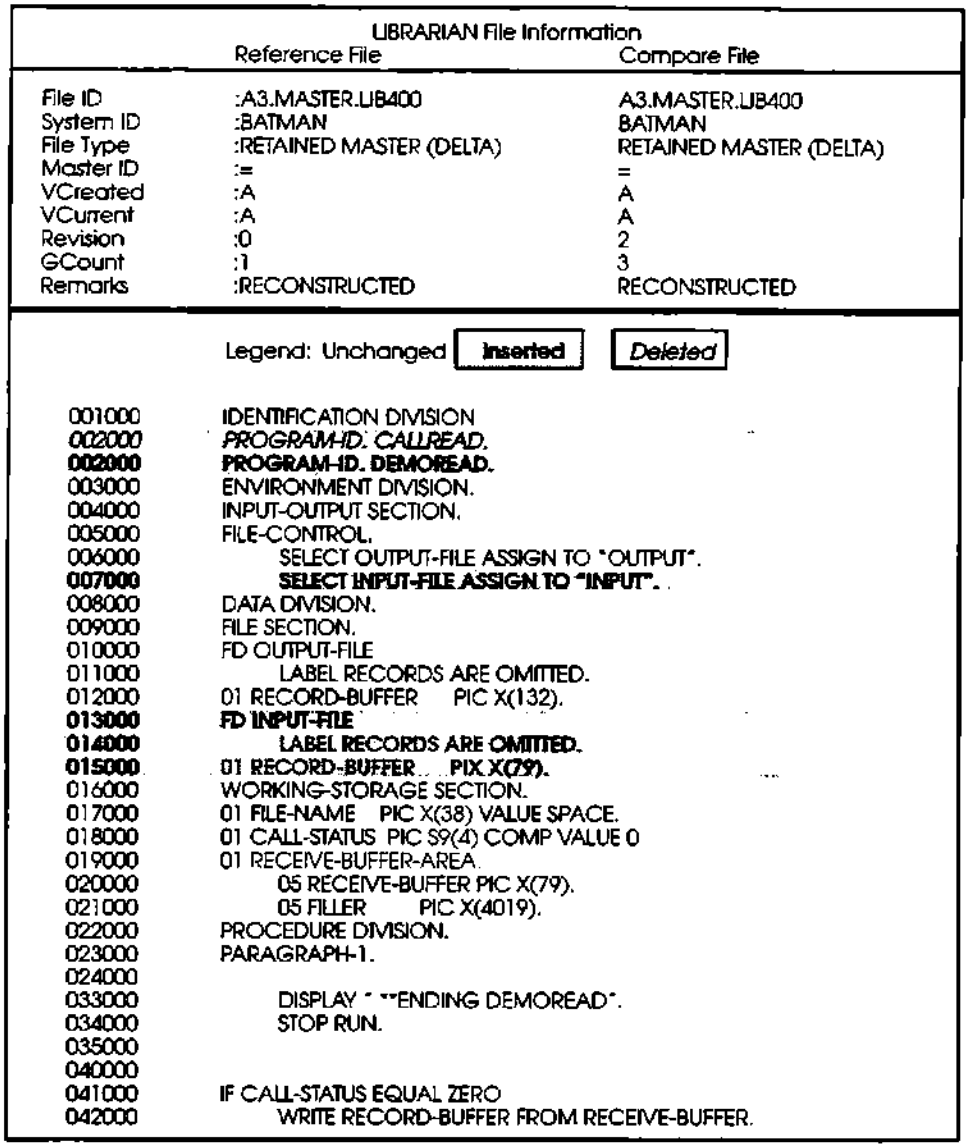
Each comparison examines the differences between a compare file and a reference file. Differences are shown as changes (insertions/deletions) to the reference file that result in the compare file. For example, when comparing a secondary to its master, the secondary is the compare file and the master is the reference file. A comparison between the two would show you changes to the master file that make it different from the secondary.

For example, the following command compares the development copies of AP files with their corresponding masters, highlighting modifications:


```
>LCOMPARE %AP AT @.@.DEVEL;MASTER
```


```
>LCOMPARE %AP AT /usr/devel/*;MASTER
```

Figure 5–2 contains an example of the online output from **LCOMPARE**.



LIBRARIAN File Information

	Reference File	Compare File
File ID	:A3.MASTER.LIB400	A3.MASTER.LIB400
System ID	:BATMAN	BATMAN
File Type	:RETAINED MASTER (DELTA)	RETAINED MASTER (DELTA)
Master ID	:=	=
VCreated	:A	A
VCurrent	:A	A
Revision	:0	2
GCount	:1	3
Remarks	:RECONSTRUCTED	RECONSTRUCTED

Legend: Unchanged | Inserted | *Deleted*

```
001000      IDENTIFICATION DIVISION
002000      PROGRAM-ID. CALLREAD.
002000      PROGRAM-ID. DEMOREAD.
003000      ENVIRONMENT DIVISION.
004000      INPUT-OUTPUT SECTION.
005000      FILE-CONTROL.
006000           SELECT OUTPUT-FILE ASSIGN TO "OUTPUT".
007000           SELECT INPUT-FILE ASSIGN TO "INPUT".
008000      DATA DIVISION.
009000      FILE SECTION.
010000      FD OUTPUT-FILE
011000           LABEL RECORDS ARE OMITTED.
012000      01 RECORD-BUFFER     PIC X(132).
013000      FD INPUT-FILE
014000           LABEL RECORDS ARE OMITTED.
015000      01 RECORD-BUFFER     PIC X(79).
016000      WORKING-STORAGE SECTION.
017000      01 FILE-NAME    PIC X(38) VALUE SPACE.
018000      01 CALL-STATUS  PIC S9(4) COMP VALUE 0
019000      01 RECEIVE-BUFFER-AREA.
020000           05 RECEIVE-BUFFER PIC X(79).
021000           05 FILLER         PIC X(4019).
022000      PROCEDURE DIVISION.
023000      PARAGRAPH-1.
024000
033000           DISPLAY " "ENDING DEMOREAD".
034000           STOP RUN.
035000
040000
041000      IF CALL-STATUS EQUAL ZERO
042000           WRITE RECORD-BUFFER FROM RECEIVE-BUFFER.
```

Figure 5–2. LCOMPARE Display

Output from **LCOMPARE** includes the filenames, file type, versions, and revisions of the files being compared. The online display uses screen enhancements to highlight changes.

If your terminal supports color enhancements, you must configure it to distinguish the types of differences between files. The standard display uses the following enhancements:

Regular Video	Indicates unchanged lines.
Inverse Video	Indicates inserted lines.
Half Inverse Video	Indicates deleted lines.

Offline listings are similar to **PRINT** listings with the **ANNOTATE** option. Deletions are shown with strikeout and insertions are shown in bold. You can change the default escape sequences for these enhancements as described in Chapter 1, "Commands" in the *LIBRARIAN/iX Reference Guide*.

Comparing Files with S/COMPARE

If S/COMPARE (proprietary product of the ALDON Computer Group) is installed on your server or MPE client, you can use the LIBRARIAN **SCOMPARE** command to access it. If you are using the menus, make sure that the compare method on the **Users...Setting** window is set to **SCOMPARE**.

SCOMPARE is similar to LIBRARIAN's own **LCOMPARE** command with the addition of many advanced options. For a description of these options, refer to Chapter 1, "Commands" in the *LIBRARIAN/iX Reference Guide*.

Because S/COMPARE only provides a 132–column display, online listings are filtered through LIBRARIAN and displayed in the manner described above for **LCOMPARE**. Offline listings are not filtered and appear in standard S/COMPARE output format.

User Filesets 6

This chapter describes user filesets and how to create and maintain them using the FMAINT module. Topics include:

- What are user filesets?
- Creating and maintaining user filesets
- Public and private user filesets
- Reviewing user fileset information
- User filesets in LIBRARIAN commands
- Project filesets
- Example

For details on command syntax and use, refer to Chapter 2, "User Fileset Commands", in the *LIBRARIAN/iX Reference Guide*.

What Are User Filesets?

User filesets allow individual users to create task-defined filesets which organize files and simplify file references in LIBRARIAN commands.

User filesets allow the programmer to create a user fileset for the files needed to complete a particular assignment and then reference the user fileset in LIBRARIAN commands. When the work is complete, the user can purge the user fileset(s).

- User filesets are similar to the master filesets that the LIBRARIAN Manager creates for the master library. These user filesets are arbitrary collections of files not necessarily related to each other by physical location.
- Whereas master filesets are defined by the LIBRARIAN Manager or Application Manager, user filesets can be defined and changed by any user. User filesets can include secondary or master files, and can also include files that LIBRARIAN is not tracking (unknown files).

The master fileset structure does not always meet each user's needs. For example, the Application Manager might create separate master filesets for application program files, application source files, and application JCL files. This arrangement groups the files logically, but a programmer might need one or two files from each master fileset for a particular assignment.

Creating and Maintaining User Filesets

User filesets can include logical and/or physical components. The user fileset name must be unique. You can create a hierarchy of user filesets depending on your own needs. Use the **FMAINT** command to access the FMAINT module from the LIBRARIAN command prompt or select **User Filesets** from the **Tools** menu. Use the FMAINT commands, or menu options, to create and maintain user filesets.

Note

In command mode, the FM> prompt indicates that you are in the FMAINT module.

Use the following commands to create a user fileset, add files to a user fileset, delete files from a fileset, and purge a user fileset:

FM>CREATE	Creates a new user fileset and optionally adds files to it at the same time.
FM>ADD	Adds files to a user fileset.
FM>DELETE	Deletes files from a user fileset.
FM>PURGE	Purges a user fileset.

User fileset hierarchies can be constructed by defining component relationships similar to master filesets. For example, if you have created several user filesets, each containing related files for a single program, you could create a single fileset containing all of the others. Using this fileset, you can move all of the files as a group. The following commands are for maintaining a hierarchy of user filesets:

FM>RELATE	Makes one fileset a component of another.
FM>SEVER	Severs the relationship between two filesets.

Public and Private User Filesets

When you create a user fileset, it is assigned a PUBLIC or a PRIVATE designation (**Default:** PUBLIC). Any user can add to or delete files from a public fileset. If the fileset creator makes the fileset private, then only that user can add files, delete files, establish component relationships, etc. Use the **MAKE** command to change the public/private attribute of a user fileset.

FM>MAKE *fileset* **PUBLIC**	Allows any general user to modify the fileset.
FM>MAKE *fileset* **PRIVATE**	Allows only the creator to modify the fileset.

Reviewing User Fileset Information

Two commands are provided to display information about user filesets. **FM>LIST** displays a list of all user filesets defined for a user, and **FM>SHOW** displays the member files and component filesets of a user fileset.

User Filesets in LIBRARIAN Commands

You can include user fileset names in LIBRARIAN commands the same way you use master fileset names, with the fileset name preceded by a percent sign (%). LIBRARIAN then authorizes each file in the fileset for the requested operation.

User filesets are restricted by the rules defined in the database. For example, if you include a file in your user fileset that you are not authorized to move, LIBRARIAN shows a violation and does not complete the move for that file.

Project Filesets

Project filesets are a special type of user fileset. A project fileset is created automatically with the same name as the project when you define that project. Files are added automatically to the project fileset as they are moved or copied with the associated project code. Although files are automatically added to project filesets, you have the option of manually altering project filesets through FMAINT and using them like any other user fileset in all respects. For more information on projects, refer to Chapter 6, "Projects", in the *LIBRARIAN/iX Administrator's Guide*.

When you use either the **CLEANDB** or **PURGE** command to remove the last master, related secondary, or retained file, the master filename will automatically be removed from the project fileset.

Additionally, if you use either the **MOVE** or **RENAME** command to remove the last master, related secondary, or retained master associated with a project, the old filename will automatically be removed and the new one will be added.

Note

Steps will automatically locate secondary file(s) in the step source location if you specify the project or project fileset.

Example

The following example shows how to create and maintain a user fileset. A programmer wants to create a user fileset to work on six files. To do this, the programmer first uses **FM>CREATE** to create the %RPT-FILES user fileset with two program files.

```
FM>CREATE RPT-FILES FROM RPT01P.OBJECT.AP,RPT02P.OBJECT.AP
```

```
FM>CREATE RPT-FILES FROM /ap/object/rpt10p,/ap/object/rpt02p
```

Then, the **FM>ADD** command is used to add the source and JCL files to the user fileset.

```
FM>ADD RPT01S.SOURCE.AP, RPT02S.SOURCE.AP TO RPT-FILES
FM>ADD RPT01J.JCL.AP, RPT02J.JCL.AP TO RPT-FILES
```

```
FM>ADD /ap/source/rpt01s, /ap/source/rpt02s TO RPT-FILES
FM>ADD /ap/jcl/rpt01j, /ap/jcl/rpt02j TO RPT-FILES
```

The %RPT-FILES user fileset now consists of the six required files. The programmer exits the FMAINT module.

```
FM>EXIT
```

The programmer checks out the required file.

```
>CHECKOUT %RPT-FILES TO =.MYGROUP
```

```
>CHECKOUT %RPT-FILES TO ./=
```

The programmer continues to use the %RPT-FILES user fileset for steps in the development route. For example:

```
>SUBMIT %RPT-FILES AT @.MYGROUP.APDEVEL
```

```
>SUBMIT %RPT-FILES AT /apdevel/mygroup/*
```

When the assigned task is completed, the programmer removes all references to the RPT-FILES user fileset from the database with **FM>PURGE**.

```
FM>PURGE RPT-FILES
```

Listfiles 7

This chapter describes listfiles and how to create and use them with LMAINT commands. Topics include:

- What are listfiles?
- Creating listfiles with LMAINT
- Maintaining listfiles
- Using listfiles

For details on command syntax and use, refer to Chapter 3, "Listfile Maintenance Commands" in the *LIBRARIAN/iX Reference Guide.*

What Are Listfiles?

Listfiles, also called *indirect files,* are files that contain a list of filenames. You can create listfiles with the LMAINT module of LIBRARIAN, with any editor, or with an application program. These listfiles can be used in LIBRARIAN commands as a way to refer to files. Listfiles can be used as indirect store lists for the MPE **:STORE** and **:RESTORE** commands, or the UNIX **tar** command, to archive files or to create distribution tapes.

Listfiles can contain filenames with wildcards. LIBRARIAN determines which files qualify when you use the listfile.

Creating Listfiles with LMAINT

You can create listfiles with any editor or you can use LMAINT commands as a convenient way to build and maintain listfiles within LIBRARIAN.

Access the LMAINT module by entering the **LMAINT** command at the LIBRARIAN prompt or by selecting the **Listfiles** option from the **Tools** menu.

Note 

In command mode, the LM> prompt indicates that you are in the LMAINT module.

Use the **LM>OUTPUT** command to create a listfile called FINFILES by typing:


```
LM>OUTPUT ABC@S.SOURCE.FINLIB TO FINFILES
LM>OUTPUT/usr/finlib/source/abc* TO finfiles
```

In the above example, LMAINT creates a listfile with a list of all tracked files satisfying the wildcard mask ABC@S.SOURCE.FINLIB (MPE) or /finlib/source/abc* (UNIX). To include untracked files in your listfile, use the **ALL** parameter. You can refer to files that you want included in a listfile in a variety of ways as described in "How to Refer to Files" in Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide.*

Additionally, you can select files based on a variety of criteria. Some of these selection criteria are described in the following sections. Chapter 3, "Listfile Maintenance Commands", in the *LIBRARIAN/iX Reference Guide* describes all possible selection options that can be used with the **LM>OUTPUT** command.

Selection by Expiration Date

You can select files for a listfile based on expiration date, using the relational operators =, <, <=, >=, or >. The following command creates the FNFIL listfile by selecting files which expire before June 30, 1993.


```
LM>OUTPUT @.@.FIN TO FNFIL ;EXPDATE<06/30/93
LM>OUTPUT /usr/fin/* TO fnfil;EXPDATE<06/30/93
```

You can use this selection feature to create a store list for archiving expired files prior to running the FLUSH utility.

Selection by File Modification Date

You can also select files based on the file modification date, using the relational operators =, <, <=, >=, or >. The following command creates a listfile of all files in the FIN account not modified since January 1, 1993:


```
LM>OUTPUT @.@.FIN TO FNFIL ;MODDATE<=01/01/93
LM>OUTPUT /usr/fin/* TO fnfil; MODDATE<=01/01/93
```

In addition, you can select files that have been modified since LIBRARIAN created them through a step or other file movement command. The following command creates a listfile of all files that were modified since they were created by a LIBRARIAN command:


```
LM>OUTPUT @.@.FIN TO FNFIL ;MODIFIED
LM>OUTPUT /usr/fin/* TO fnfil;MODIFIED
```

You can compare the modification timestamp of files to the timestamp of a specific file. For example, you could create a listfile including the names of all files that were changed since the last time the listfile was created.

The following example uses the **RESETONZERO** parameter to empty the listfile if no files qualify using the selection criteria:

```
LM>OUTPUT @.@. FIN TO FNFIL;MODDATE>TIMESTAMP(FINFILES) &
;RESETONZERO
```

```
LM>OUTPUT /usr/fin/* TO fnfil;MODDATE>TIMESTAMP(finfiles)&
;RESETONZERO
```

Selection by Simulating a LIBRARIAN Step

You can use LMAINT to simulate execution of a defined step by invoking the USE option of the **OUTPUT** command. With the **USE** option, LIBRARIAN identifies the step destination for each selected file for the step and places that filename in the listfile without performing the step. The following command simulates a defined checkin step for all files in the project fileset SR1234:

```
LM>OUTPUT %SR1234 ;USE CHECKIN
```

Since this is a simulation of a step, it has no impact on revisions or versions.

The **SIMULATE** option is similar to the **USE** option, with the exception that it only includes destinations for authorized files. The **AUTHORIZE** option is also similar to the **USE** option, with the exception that it only includes files that would be authorized for step (i.e., source locations).

Maintaining Listfiles

You can create and edit listfiles outside of the LMAINT module by using any editor (e.g., **vi** in UNIX). In addition, other LMAINT commands allow you to add documentation, sort, or modify the contents of a listfile.

The following LMAINT commands are available from MPE only.

You can use the **LM>SORT** command to sort a listfile and, optionally, eliminate duplicate filenames.

The **LM>EDIT** command allows you to edit the contents of a listfile using EDIT/3000.

The **LM>DOCUMENT** command allows you to add text or edit a maximum of 750 lines of notes for the listfile with the EDIT/3000.

If you want to append filenames to a listfile, use the **LM>ALTER** command to toggle the mode for that particular file to append. Subsequent output to the file with the **LM>OUTPUT** command is appended to the existing file. For example:

```
LM>ALTER FINFILES ;APPEND
```

The **LM>REPORT** command allows you to report documentation notes for the listfile, summary information (i.e., creation and modification dates), and filenames contained in the listfile.

The **LM>LIST** command lists the filenames contained in a listfile.

Using Listfiles

You can use listfiles in any LIBRARIAN command as a way to refer to files. Listfiles are prefixed by ^ or ! , as shown in the following examples:

```
>COPY ^MYFILES TO =.=.TESTACCT
>MOVE !APFILES TO TESTSYS:=.=.=
>CHECKOUT ^NEWSRCE
```

```
>COPY ^myfiles TO /usr/test/=
>MOVE ^apfiles TO TESTSYS:=
>CHECKOUT ^newsrce
```

Indirect Store Lists

Listfiles can be used with many utilities, including the MPE **:STORE** command for indirect store lists. Use a listfile in the **STORE** command by preceding the command with ! , as shown in the following example:

```
:FILE T; DEV=TAPE
:STORE  MYFILES; *T; SHOW
```

For more information on indirect **STORE** files, refer to the *MPE Command Reference Manual.*

Archiving Applications with Listfiles

Use LMAINT to facilitate the selection (and compression, if desired) of files to be archived and stored to tape. You can use the same selection list to store obsolete files and then purge the files from disk.

The following procedure describes how to archive files that LIBRARIAN is tracking, as well as files not being tracked. Some aspects of the archiving process are not available with files that are not being tracked by LIBRARIAN.

Use LMAINT to write the filenames to a text file. For example:

```
>LMAINT
LM>
```

Use the **LM>OUTPUT** command to select files and list them to a file. For example:

```
LM>OUTPUT AB@.SOURCE.INVTRY TO APR1589.STORE;ALL
```

```
LM>OUTPUT /invtry/source/ab* TO ./store/apr1589;ALL
```

This command creates a text file which contains all files described by the mask, regardless of whether or not these files are tracked in the LIBRARIAN database (ALL parameter).

The following procedure describes a convenient way to compress and archive a retained version of an application:

1. Access LIBRARIAN by typing:

```
:LIB

>USER userid
Password?
Press F2 to switch to command mode.
```

2. Copy the retained version to a temporary archive area by typing:

```
>COPY VERSID OF %APPL TO .=. =.ARCHIVE;OLDNAME;&
>COMPRESS
```

```
>COPY VERSID OF %APPL TO ./archive/=;OLDNAME;&
>COMPRESS
```

3. Access the LMAINT module by typing:

```
>LMAINT
```

4. Create the STORE listfile with the names of all files just copied by typing:

```
LM>OUTPUT * TO STORELIST
LM>OUTPUT ** TO STORELIST
```

5. Exit LMAINT by typing:

```
LM>EXIT
```

6. Store the files to tape by typing:

```
>STORE ^STORELIST
>cpio -o<STORELIST
```

7. Purge the files that you archived to tape by typing:

```
>PURGE ^STORELIST
```

The result is a complete archive tape of an application version in compressed form. Step 7 does not necessarily purge all files which have been stored to tape, since some version files, if unchanged, remain members of the application in later releases.

8. Make the REL2 version obsolete by using the **VERSION** command and the **OBSOLETE** parameter. For example:

```
>VERSION APPL; ID=VERSID ;OBSOLETE
```

9. Run the FLUSH utility to remove the obsoleted version.

```
>FLUSH
```

Rebuilding Applications with MAKE 8

MPE only

The MAKE facility automatically rebuilds/recompiles changed components of an application based on a set of user–defined rules. This chapter describes MAKE and how it works. Topics in this chapter include:

- Why Use MAKE
- How MAKE Works
- Creating MAKE files
- Executing MAKE

Why Use MAKE?

LIBRARIAN's MAKE facility is modeled after the UNIX program, **make**. MAKE helps keep applications up–to–date by rebuilding or recompiling only the changed parts of the application.

Large applications can have hundreds of modules, each of which depends upon other modules. Manually tracking all the pieces of an application is a time consuming and tedious task. Moreover, forgetting to recompile a module that has changed — or that depends on something you changed — can lead to serious problems. On the other hand, recompiling everything is a waste of time and resources.

MAKE helps maintain any application by:

- centralizing rules for rebuilding application components,
- accommodating new modules easily,
- providing variables and generic rules to eliminate redundancy, and
- eliminating the need for compile jobs/scripts.

You provide MAKE with a set of rules describing how to rebuild an application's components (targets) when any associated dependencies have changed. MAKE looks at these dependency rules, compares the modification timestamps between target and dependency files, and performs the necessary tasks to create an up–to–date version. MAKE never performs more work than is necessary to bring an application up–to–date.

An example of a dependency relationship is that between executable (target) and source code (dependency). When a source file is modified the corresponding object code needs to be rebuilt, typically through a set of compile and link commands. MAKE compares the timestamp of the executable to the timestamp of the source code and performs a compile and link, if necessary.

In addition to building programs, MAKE can also be used to run automated test suites, extract and process data, rebuild documentation, generate reports when new data is available, etc. For example, MAKE could be instructed to launch a test script whenever the program it tests changes. In this case, MAKE retests only those parts of the application that have changed. MAKE applies to any situation where there are timestamp dependency relationships between files, and a known set of commands to execute when a dependency changes.

MAKE has several advantages over using jobs or scripts to rebuild each component of an application:

- MAKE removes the burden on users to remember what has changed and what components depend on those changes.
- Variables in MAKE allow you to combine similar rules in a generic way, requiring fewer instructions and files to perform the same task.
- MAKE recompiles only the components whose dependencies have changed; jobs, however, require manual timestamp comparisons or knowledge about what has changed.
- MAKE provides generic rules that allow you to add new components without changing the makefile. With jobstreams, you need to alter the jobs or add new jobs each time a new component is introduced.

How MAKE Works

Components built through MAKE are called *targets*. Targets include applications, programs, object files, and libraries; that is, anything you can build. For an application to be up–to–date, its executables need to be up–to–date. For the executables to be up–to–date, the linker libraries need to be up–to–date, etc.

MAKE keeps applications up–to–date in the following way:

- Reads a file (called a *makefile*) that contains a set of rules. This file includes target components of your application, associated dependencies, and the commands necessary to bring each target up–to–date. Targets can depend on other targets in a hierarchical fashion.

- Compares the modification timestamp of each target against its dependencies. If the target is *older* than any of its dependencies, MAKE generates the series of commands required to rebuild the target. If that target, in turn, happens to be a dependency of another target, then it, too, will be rebuilt. Strict ordering is enforced so that components at the lowest level of the hierarchy are built first. If a target does not exist, it is always rebuilt.
- Streams or schedules a job to execute the commands necessary to bring the entire application up–to–date.

To illustrate how MAKE works, consider a sample application which contains four modules written in COBOL. To produce an up–to–date version of the application, you must compile each of the source modules (MOD1 - MOD4) into respective object files (MOD1OBJ – MOD4OBJ), and then link them into an executable program (MYPROG). To ensure that the program is up–to–date, you must recompile the modules that have changed since the last time you generated the object file.

MAKE provides an automated method for identifying and recompiling changed components. Figure 8–1 illustrates how MAKE handles the recompilation of changed source code modules.

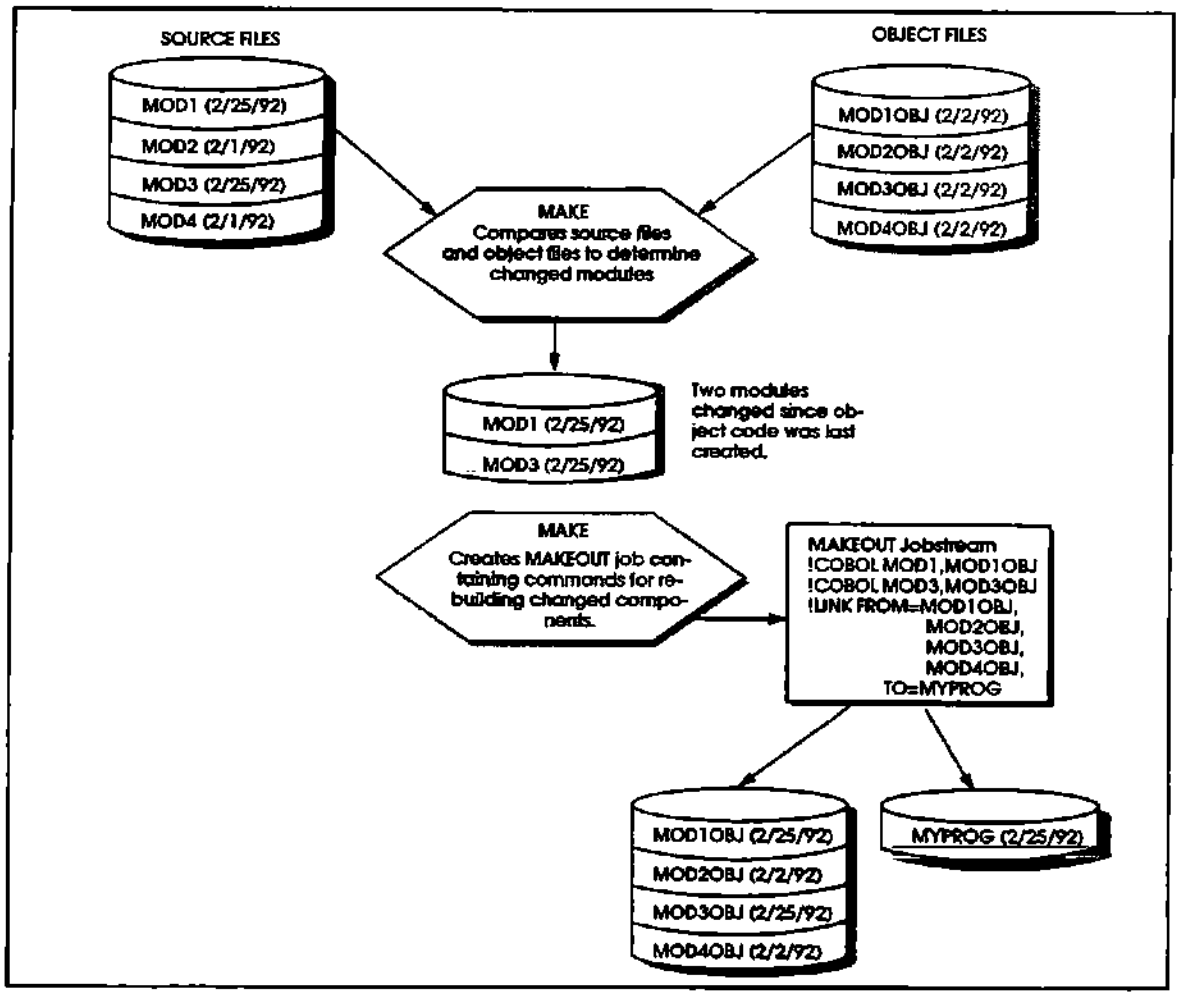


Figure 8–1. Example of a MAKE Operation

Figure 8–1 shows that the object files were last updated on February 2nd. On February 25th, the source files, MOD1 and MOD3, were modified. The makefile states that any source files modified since the object code was last compiled need to be recompiled using the command:

```
! COBOL $< , $*
```

($ < and $* are examples of MAKE's powerful variables. These variables get replaced with the name of the changed dependency — in this case, the files MOD1 and MOD3 and associated target names, respectively). The program file is then linked using the command:

```
!LINK FROM=MOD1OBJ,MOD2OBJ,MOD3OBJ,MOD4OBJ;TO=MYPROG
```

MAKE streams a job (called MAKEOUT) with these commands to bring the changed components up-to-date.

Defining the Dependency Tree

Before creating the makefile, determine application file dependencies. You might find it useful to map out the dependency tree for the application before actually creating the makefile.

To begin building the dependency tree, determine the ultimate target for your application. This target is typically an executable program file or set of application programs. For example, the target of the application in the previous section was to bring the executable program, MYPROG, up–to–date.

Direct dependencies of the highest target are listed underneath the primary target. In the MYPROG example, the executable file depends upon the object files. The object files, in turn, depend upon each of the source files.

Figure 8–2 illustrates the dependency tree for the MYPROG application.

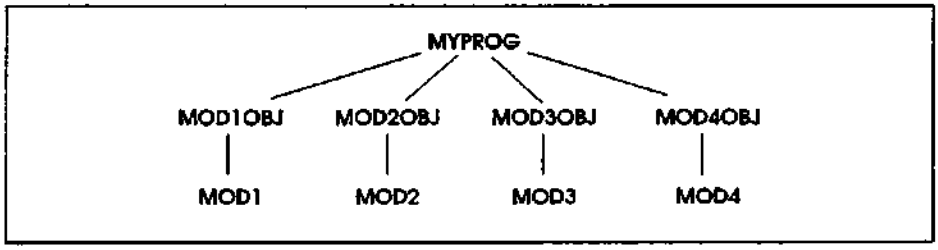


Figure 8–2. Dependency Tree for MYPROG

Creating Makefiles

A makefile is simply a text file that contains one or more MAKE rules. Each rule defines a specific or generic target/dependency relationship and the commands required to rebuild the target from the dependencies. MAKE can handle a variety of tasks in developing, testing, and releasing applications. Therefore, any command, or series of commands, is valid. Create and maintain this file in the editor of your choice. Although *makefile* is the default name MAKE uses for this file, any name is acceptable.

Conventions

When creating a makefile, adhere to the following conventions:

- Put a blank line between rules.
- Use the slash (\ ) as a line continuation character.
- When listing targets and dependencies use a minimum of one space between filenames. Do not use commas!

Comments

Comments are written on separate lines and can appear anywhere in a makefile. The first non-blank character in a comment line must be a #. For example:

```
#This is a comment.
```

Comments that begin with #NOTE are treated in a special way at runtime. Use the **ECHO** option with MAKE (see *Executing MAKE* below), and these comments are displayed on screen as the makefile is processed. For example, suppose your makefile had this comment:

```
#NOTE Processing report rules...
```

When MAKE processes this file, the following line appears.

```
Processing report rules...
```

Comments that begin with #OPTION followed by an option list are also treated in a special way at runtime. For example:

```
#OPTION SHOW ECHO
```

The option list can include any MAKE parameters (**SHOW**, **ECHO**, **NOMAKE**, **ALL**, etc.) as described in Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide.*

Rules

Rules are statements that inform MAKE about file dependencies and what action to perform when dependencies change. The dependency tree described in the previous section is a graphic representation of the rules in a makefile. Each rule has the general format:

```
<target list> : <dependency list>
    <commands>
```

where,

target list — Specifies the name(s) of the target(s) that must be rebuilt if any file in the dependency list has changed (i.e., the timestamp of the target is older than the dependency).

Targets can be file names, variable expressions, or dummy names. If a dummy name is used that does not correspond to an existing file, the *commands* (see below) are always performed.

dependency list — Specifies the names of the dependencies of the target. If any file in the dependency list has changed, the *commands* given for the rule are performed.

commands Specifies the operating system command(s) you wish to execute if the target is older than any of its dependencies. Any number of commands can be issued for each target/dependency list.

Commands are placed in a jobfile called MAKEOUT, by default. Thus, all commands in the action section must conform to standard JCL conventions, including prefixing commands with a job character (e.g., !).

Note Commands used to rebuild may be entered into any column beyond Column 1.

The following section describes how to transform the MYPROG dependency tree into a makefile.

Example 1: The Basics

As an example of creating a makefile, consider MYPROG and its associated source files. The dependency tree for MYPROG (shown in Figure 8–2) has three levels: the program file, the object files, and the source files. Traversing the tree from top to bottom expresses the rules in the makefile. The MYPROG dependency tree illustrates two rules:

- The MYPROG file target depends upon the object files MOD1OBJ, MOD2OBJ, MOD3OBJ, and MOD4OBJ dependencies
- The object files, in turn, depend upon the corresponding source files.

Figure 8–3 is an example of a makefile for MYPROG.

```
# Build the MYPROG program file
MYPROG : MOD1OBJ MOD2OBJ MOD3OBJ MOD4OBJ
                :>!JOB MAKEPROG,MGR.MYACCT/PASSWORD
                !LINK FROM=MOD1OBJ,MOD2OBJ,MOD3OBJ,MOD4OBJ;TO=MYPROG

# Build Object MOD1OBJ
MOD10 : MOD1
                !COBOL MOD1,MOD1OBJ
# Build Object MOD2OBJ
MOD20 : MOD2
                !COBOL MOD2,MOD2OBJ
# Build Object MOD3OBJ
MOD30 : MOD3
                !COBOL MOD3,MOD3OBJ
# Build Object MOD4OBJ
MOD40 : MOD4
                !COBOL MOD4,MOD4OBJ
```

Figure 8–3. Makefile for MYPROG Example

How MAKE Interprets the MAKEFILE

In Figure 8–3 the target MYPROG has four dependencies. The first dependency, MOD1OBJ, is a target of another rule and must be evaluated first to determine if it needs to be rebuilt . The target MOD1OBJ has one dependency, MOD1. MAKE checks if MOD1 is out–of–date and issues a command to rebuild MOD1OBJ. Similarly, the other three dependencies of MYPROG are evaluated and then MAKE returns to the first rule to rebuild target MYPROG, if necessary.

The makefile in Figure 8–3 can be made much shorter and more effective, using predefined variables in a generic rule, as shown in Figure 8–4.

```
# Build the MYPROG program file
MYPROG : $(MOD#OBJ)
        :>!JOB MAKEPROG,MGR.MYACCT/FOOBAR
        !LINK FROM=MOD1OBJ,MOD2OBJ,MOD3OBJ,MOD4OBJ;TO=MYPROG

# Create the object file by compiling the source files
MOD#OBJ :- =—
        !COBOL $<, $*
```

Figure 8–4. Makefile for MYPROG Example

In this case, the dependencies, MOD#OBJ are determined from a LISTF. These files qualify as targets in the second generic rule, and must be evaluated first. MAKE executes the commands of the second rule for any source files that are out-of-date, and then returns to the first rule to rebuild the MYPROG file using the **LINK** command.

MYPROG Rule

The makefile begins with a comment to inform us of the purpose of the makefile. The first rule in this makefile specifies the ultimate target — to produce the MYPROG executable program. This standard rule means, *If program file MYPROG is older than the object files on which it depends, proceed with the commands below to rebuild the program file.*

The dependency list in this rule is generated by using the LISTF variable ($[ ]). The LISTF variable finds files that match the pattern given between the brackets and then substitutes the names of any files found. In our example, MOD1OBJ MOD2OBJ MOD3OBJ MOD4OBJ replaces $[MOD#OBJ] at runtime.

Note

To accommodate new source files, generate a list of object dependencies by doing a **LISTF** in the source area as follows: $[MOD#]"@OBJ" This applies the edit mask "@OBJ" to each source file found via **LISTF**. Since object for new source files will not exist, MAKE automatically builds them.

The first command in the first rule is a special command which specifies the login for the MAKEOUT job. The job command must be the first command in a rule, with the special prefix, ":>". In this example, if any targets are out-of-date, the MAKEOUT job created by MAKE logs on to MGR.MYACCT using the password FOOBAR.

Note

The job statement should be specified for the first rule and for any rule that could be an entry point into the makefile. MAKE allows you to evaluate any target in the makefile, but by default it is the first target. For more information about executing MAKE with target entry points, refer to *"Executing MAKE"* later in this chapter.

Following the job command is the actual command used to rebuild the program file from the object file.

MOD#OBJ Rule

The second rule in this makefile illustrates the power of MAKE's variables in conjunction with generic rules (by default, variables are prefixed by the dollar sign ($), but the next section describes how to change this prefix). The purpose of this rule is to rebuild any object file whose source files have changed. In our example, we have only four object files, but this rule is valid for any number of object files that follow the naming convention.

This is a generic rule indicated by the ":–" delimiter between target and dependency. Dependencies are determined in this case by applying an edit mask to the target being evaluated. In this case, the associated dependency is determined by removing the last three characters (=---).

In the commands of the second rule, two more variables ($< and $*) are used. The $< variable is replaced with the name of the current dependency and the $* variable is replaced with the name of the current target exactly as entered in the rule (another variable, $@, represents the fully qualified target name with account and group). Thus, the command that is written to the MAKEOUT job when MOD3 has changed is the following.

```
!COBOL MOD3,MOD3OBJ
```

Example 2: A Comprehensive Illustration

The example in Figure 8–4 illustrates how easy it is to use MAKE for rebuilding a simple software application with any number of component modules. Now examine a more comprehensive example that really takes advantage of the power of MAKE . You will find that once you get to know the MAKE syntax, even complicated applications can be managed with a few simple MAKE rules.

This example consists of a financial software application. All files are contained in the DEVEL account, but the source files for some library routines, written in Pascal, are kept in the PASCAL group and the source files for the remainder of the application, written in COBOL, are kept in the COBOL group. The object code for these routines is placed in an RL called FINRL. The application also has an outer block module written in C, called FINSRC. The corresponding object for FINSRC is FINOBJ. Our finance application program FINP is created by linking FINOBJ and FINRL.

The application also has a set of associated reports that must be built. The source code for reports resides in the RSOURCE group, and each source file ends in the letter S. The compiled reports need to be placed in the PROG group and the names of the compiled reports are the same as the source files, except the last character (S) is removed. Figure 8–5 shows the dependency tree for this application, and Figure 8–6 shows the makefile used to build the FINANCE application.

```
    $ = %
    ACCT = DEVEL
    PASS = FOOBAR

    # Build the Finance application
1.  FINANCE : FINP FINRPTS
                    :>!JOB MAKEFIN,MGR.%(ACCT)/%(PASS)

    # Link FINP
2.  FINP : FINRL FINOBJ
                    !LINK FROM=FINOBJ;TO=FINP;RL=FINRL

    # Compile FINSRC - C source
3.  FINOBJ : FINSRC
                    !CCRL %<,%@,$NULL

    # Compile COBOL library code and update FINRL
4.  FINRL :: %[@.COBOL]
                    (
                    !COBOL %<,,$NULL
                    !IF JCW < FATAL THEN
                                !LINKEDIT
                                RL FINRL
                                PURGERL MODULE=%<"="
                                ADDRL $OLDPASS
                                EXIT
                    !ENDIF
                    )
5.  # Compile PASCAL library code and update FINRL
    FINRL :: %[@.PASCAL]
                    (
                    !PASXL %<,,$NULL
                    !IF JCW < FATAL THEN
                                !LINKEDIT
                                RL FINRL
                                PURGERL MODULE=%<"="
                                ADDRL $OLDPASS
                                EXIT
                    !ENDIF
                    )
    # Compile and link reports
6.  FINRPTS : %[@S.RSOURCE]"=-.PROG"
                    :>!JOB MAKERPT,MGR.%(ACCT)/%(PASS)
7.  @.PROG :- @S.RSOURCE
                    (
                    !COBOL %<,,$NULL
                    !CONTINUE
                    !IF JCW < FATAL THEN
                    !LINK FROM=$OLDPASS;TO=%@
                    !ENDIF
                    )
```

Figure 8–5. Dependency Tree for the FINANCE Application

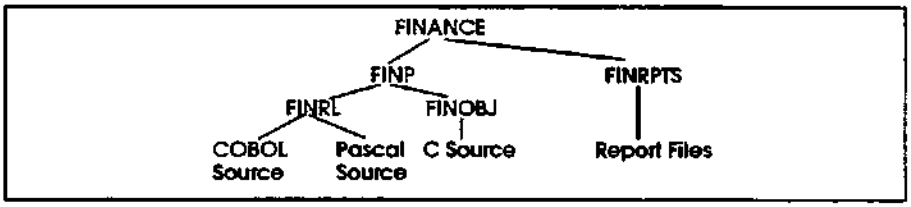


Figure 8–6. Makefile for the FINANCE Application

The new concepts introduced with this makefile are dummy targets, user-defined variables, iterative command processing, job card placement, edit masks, and rule delimiters.

Dummy Targets

Dummy targets are target names that do not correspond to any existing file. Dummy targets are always built. The FINANCE and FINRPTS targets in rules 1 and 6 of the makefile are examples of dummy targets.

User-Defined Variables

The first three lines of this makefile define variables for use in the rest of the file. Variable definitions have the following format:

variable_name = substitution_text

The substitution text replaces every reference to the variable in the makefile. The first user-defined variable in Figure 8–6 is special — it causes the percent sign (%) to be used as the variable prefix rather than the default dollar sign ($). This variable is necessary in order to avoid confusion between system defined filenames, such as $OLDPASS or $NULL, and variables in your makefile.

The other user-defined variables (ACCT and PASS) are useful because there are several places where the account name and password are used. Since passwords change frequently, you only need to change the value of the PASS variable once and the correct password is replaced in the appropriate locations.

A user-defined variable is referenced in the same manner as a predefined variable. The variable prefix (in this example, a percent sign [%]) must precede the variable's name. If a user-defined variable name contains *more* than *one* character, the variable name must be enclosed in parentheses. For example, notice the parentheses in the reference to the %(ACCT) and %(PASS) variables in the job login in the first and fifth rules.

Iterative Command Processing

One important point to notice in several of these rules is the use of braces ({ }) before and after the commands of the rule. Braces instruct make to iterate the commands between them for each changed dependency (the current changed dependency name is substituted for the $< macro during each iteration). If no braces are placed around the command, the command is performed once for the first dependency in the dependency list, regardless of which dependency has changed.

The FINANCE example shows simple iterative command processing. In the example, braces are used to indicate that the commands are to be performed once for each out-of-date dependency. If a target needs to be rebuilt, commands can be iterated using any of the following criteria in any combination:

- once for every changed dependency (braces)
- once for each dependency (brackets)
- once for first dependency (no braces/brackets)

Let us examine another example in which iterative command processing is useful. In this example, we have a screen driver that depends on the source files for the individual screens.

```
$=%
SCREEN.PROG: %[@.SCREENS]
        {
        !PASCAL %<,SCREEN.OBJ,$NULL
        }
        !LINK FROM= SCREEN.OBJ;TO=%@
```

The block of commands between the braces are repeated once for each dependency that has changed. The dependency variable value is dynamically altered to reflect the current dependency at each iteration.

Notice how **!LINK** is located outside of the iterative block. Since no braces/brackets surround this command, it is executed once after all of the changed source files have been recompiled.

Alternatively, you can create a block of commands that is executed for every dependency, regardless of which dependency in the list has changed, as in the following example:

```
SCREEN.PROG : $[@.INCLUDE]
        [
        !FILE %< = %<"=.EXTERNAL"
        ]
        {
        !RESET %<
        }
        !PASCAL SCREEN.SOURCE
```

This example instructs MAKE to issue a file equation for all includes to point to an associated external declaration. Then, it resets the file equations for only those that have changed.

Job Card Placement

By default, MAKE begins by evaluating the first rule of the makefile and continues processing the file sequentially. However, you can specify any target in the makefile for MAKE to process. Since MAKE can potentially enter the makefile from any target, you must define job cards wherever this is likely.

In this example, you might want to rebuild the report programs without rebuilding the entire application. To accomplish this you would issue the following command to invoke MAKE:

```
:MAKE MAKEFIN,FINRPTS
```

MAKE then enters the makefile at the fifth rule and processes that rule and related rules only, ignoring all other rules.

The job card must be the first command in a rule. MAKE uses the first job card it encounters as the login for the MAKEOUT job. Any subsequent job cards encountered by MAKE are ignored.

Edit Masks

Edit masks are used throughout a makefile for two purposes:

- to determine the dependency of a target in a generic rule
- to edit the value of a predefined file variable or file names returned by the LISTF variable.

Edit masks use the special characters, @, =, ?, and −, as well as literal characters. Enclose edit masks in quotes immediately following a file variable reference to temporarily modify its value. For detailed information about each edit mask character, refer to "Edit Masks" at the beginning of Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide*.

In rules 4 and 5, an edit mask is used to extract the filename (without group and account) from the current dependency variable value.

```
PURGERL MODULE = %< "="
```

In rule 6, an edit mask is used with the LISTF variable to create a list of report program names in the PROG group by removing the last character from the source file names in the LISTF result.

```
FINRPTS : % [@S.RSOURCE] "= − .PROG"
```

In rule 7, an edit mask is used as the dependency in a generic rule, so that MAKE can determine the dependency of a report target it evaluates (no quotes are required in this case).

```
@.PROG :− @S.RSOURCE
```

Standard (Specific) Rules

There are several kinds of rules that MAKE recognizes based on the delimiter between the target list and the dependency list. Rules one through six are standard rules delimited by a single colon (:) or double colon (::). Rule 7 uses the (:−) colon–dash delimiter to define a generic rule.

The single colon and double colon delimiters are closely related. They are both used for specific rules where the target list is a specific list of filenames and the dependency list does not use edit masks. The difference between the two delimiters is how rules with targets of the same name are treated.

- The *single colon delimiter* (:) causes all rules with targets of the same name to be combined as though they were one rule (i.e., dependencies and commands are combined).
- The *double colon delimiter* (::) causes each rule to be evaluated independently, and only the commands of the rule whose dependencies have changed are executed.

As an example of the difference between the single colon and double colon delimiters, consider rules 4 and 5. Both rules have the same target, FINRL. Rule 4 states that all out-of-date modules in the COBOL group should be rebuilt using the COBOL command. Rule 5 states that all out-of-date modules in the PASCAL group should be rebuilt using the PASCAL command. If a single colon were used to delimit these rules, the two commands would be combined into one rule and the following commands would be issued if the ABCCOB5 module is out-of-date:

```
!COBOL ABCCOB5,,$NULL
!PASCAL ABCCOB5,,$NULL
```

This is clearly not desired. Therefore, each rule must be treated independently (i.e., if a module in the COBOL group is out-of-date, perform one command, and if a module in the PASCAL group is out-of-date, perform a different command). In the case of rules 4 and 5, the double colon delimiter guarantees the independence of the two rules.

Generic Rules

Specifying wildcard characters in a target makes the rule generic so that a single rule can apply to any number of targets that match. As MAKE examines a makefile for dependencies that are themselves targets, MAKE checks generic target names for a match. The :- and := delimiters are used to specify a wildcard pattern for the target name and a corresponding edit mask as the dependency name.

The := delimiter can be used only when the target and dependency names are the same, but with different suffixes (e.g., target ABC199M and dependency ABC199S would be covered by the rule).

The :- delimiter is more flexible than :=. The :- causes MAKE to determine the corresponding dependency name from the edit mask. For example, consider the last rule in the FINANCE makefile:

```
@.PROG :- @S.RSOURCE
    ...
```

This rule states that a dependency derives its name from the target being evaluated. Thus, an S is added to the target name and the RSOURCE group is added to derive the dependency name. If the target file were RDV10.PROG, the dependency would be RDV10S.RSOURCE.

MAKE also supports multiple dependencies using edit masks in this type of rule. For example,

```
@.COMP  :-   =S.SOURCE  =F.FORMS  RL.COMP
```

In this rule, each program file in the COMP group is dependent on its source file, a forms file in the FORMS group, and an RL file in the COMP group.

Implicit Rules

A variation of the standard rule is the *implicit rule* (often called the *UNIX generic rule*). This type of rule is used when both the target and dependency have the same name, but reside in different groups. In this construct, there is no dependency list to the right of the rule delimiter. For example, the following rule states that MAKE should evaluate all targets in the OBJECT group against all files of the same name in the SOURCE group.

```
. SOURCE.OBJECT :
```

A target named ABC100.OBJECT would need to be rebuilt if ABC100.SOURCE has changed.

Automatic Search for Include Files

When a plus sign(+) is entered after a filename, files are scanned for references to include files. The file and all its includes are taken as dependencies. For example:

```
filename+  or  $[@src+]  or  $[!listfile+]
```

Currently, MAKE supports this feature for COBOL, C and PASCAL.

Listfiles in Generic Rules

Generic rules can refer to generic listfile names. The listfile name is determined from the target name using an edit mask (similar to the way generic dependencies are determined.) For example:

```
XX@O :- $[!=L]
```

returns a dependency list with the filenames in a listfile with the same name as the target, except for the last letter which is "L".

LISTF Variable Exclusions

The LISTF variable supports exclusions. For example:

```
$[A@-A1-A2-A3]
```

excludes A1, A2, and A3 from A@.

Special MAKE Variables

In addition to the variables already mentioned in this chapter, four special variables are available. These variables include:

- STREAM
- SCHEDULE
- ACCOUNT
- GROUP
- ALTPATH
- EXCLUDE
- COPYMEM
- Prompt variables
- System variables

STREAM

You can optionally specify parameters for the MPE **:STREAM** command to be issued when MAKE streams the MAKEOUT job. When the STREAM variable is defined, its value is passed as a parameter list to STREAM. For example, if the following macro is used anywhere in the makefile, then MPE launches the job at 5:00 p.m.

```
STREAM = AT=17:00
```

For more information on STREAM, Refer to the *MPE Commands Reference Manual*.

SCHEDULE

For users who have scheduling or streamer programs, MAKE recognizes the SCHEDULE variable. If the user defines a variable named SCHEDULE anywhere in the makefile, then MAKE expects its value to be the name of the scheduler program. MAKE runs this program and passes the name of the MAKE jobstream via the info string, instead of streaming the file. The program name may optionally have a slash (/) at the end, followed by S, P, or G corresponding to the **LIB=x** parameter that the scheduler program requires.

For example, if you define the following variable in a makefile, then MAKE would run **STREAMER.COMP.EXPRESS** with a **LIB=G** parameter. MAKE passes the name of the MAKE jobstream in the info string rather than streaming the MAKE command file directly to MPE.

```
SCHEDULE = STREAMER.COMP.EXPRESS/G
```

Both STREAM and SCHEDULE

If you define both the STREAM and SCHEDULE variables, MAKE invokes the scheduler and appends the stream options to the info string, with a semicolon delimiter. The EXPRESS **STREAMER** command, for example, implements the same options as the MPE **:STREAM** command. This also provides a means of specifying additional scheduling parameters.

ACCOUNT

If you run MAKE outside of the account where the files to be evaluated reside, you can use the special ACCOUNT variable to set the account globally. With this variable, you only need to qualify your target and dependency filenames up to the group level in the makefile. For example:

```
ACCOUNT = QAACCT
```

If you specify the ACCOUNT variable in the makefile, you can only specify filenames up to the group level, since the ACCOUNT variable appends the account name to all filenames in the makefile.

GROUP

If you run MAKE outside of the group where the files to be evaluated reside, you can use the special GROUP variable to set the group globally. For example:

```
GROUP = MAKEGRP
```

If you define the GROUP variable in the makefile, only specify filenames, since MAKE appends the group name to all filenames in the makefile.

ALTPATH

The ALTPATH variable causes MAKE to automatically search an alternate account when a dependency is not found in the default account defined by the ACCOUNT variable or logon account. You should set the ALTPATH variable to the account you want MAKE to search as an alternate for dependencies. For example:

```
ACCOUNT=ABCDEV
ALTPATH=ABCLIB

ABC : $[@.PROG]
      :>!JOB.....

ABC1000P.PROG : ABC1000S.SOURCE
      !rebuild statements...

@P.PROG :- =S.SOURCE
      !rebuild statements...
```

If the dependency for a target does not exist in the same account, MAKE searches for the same *file.group* in the ALTPATH account. For example, if ABC3000P.PROG is found in the account ABCDEV, but ABC3000S.SOURCE.ABCDEV does not exist, MAKE searches for ABC3000S.SOURCE.ABCLIB. If MAKE finds the dependent file in the ALTPATH account, it uses that file as the dependency. All other MAKE logic remains the same.

This variable is useful when compiling in an account that only has modified source files and not the entire library. Using ALTPATH, you can issue file equations using iterative command processing for *all* dependencies to point to files in the library that are not in the account where the compile is taking place. Then use iterative command processing for *changed* dependencies to reset the appropriate file equations.

For example:

```
ACCOUNT = ABCDEV
ALTPATH = ABCLIB

ABC : $[@.PROG]
[
!FILE %< = %<<
]
{
! RESET %<
}
!rebuild commands...
```

Note the special variable %<<, which means the dependency name qualified with the current account. %< always refers to changed dependencies which could be in either account.

EXCLUDE

The EXCLUDE variable can be used to exclude delta files and generation files from a LISTF variable.

For example:

```
EXCLUDE = D#######.@.@ G#######.@.@
```

excludes D#######.@.@ and G#######.@.@ files from all dependencies lists that use the LISTF variable.

COPYMEM

The COPYMEM variable is used in conjunction with the MAKE automatic dependency scan feature to indicate that copylib members are stored as individual files in [GROUP[.ACCOUNT]] as opposed to using the COPYLIB file itself as the dependency. For example:

```
COPYLIB=MYGROUP
COPYLIB=MYGROUP.MYACCT
```

Prompts

You can prompt the user for the value of a variable. Prompt variables have the general format:

```
${ prompt_text }
```

You can use this type of variable prompt for filenames and passwords. For example,

```
:>!JOB MAKEPROG, MGR.MYACCT/${Password:}
```

The variable above causes MAKE to prompt the user with "Password:" when MAKE is run. The text that the user enters at a prompt is inserted in the makefile.

You can also use prompt variables to allow a user to enter a list of files to build.

For example:

```
MYBUILD : ${List files to build:}
        {
        !COBOL $<,,$NULL
        }
        . . .
```

The use of the prompt variable above allows you to provide dependencies at runtime.

System Variables

You can substitute the value of MPE/iX system variables in MAKE files with the following syntax:

```
$(!system_variable)
```

Executing MAKE

Execute MAKE by using the LIBRARIAN **MAKE** command or selecting **Make** from the **Tools** menu. With the **MAKE** command you supply the makefile name and, optionally, the target entry point, a listing filename, and a job filename.

The first target in a makefile is the default target that MAKE builds. You can override this default and instruct MAKE to enter the makefile from any target you choose. If you plan to do this, be sure the target you choose as an entry point into the makefile has a job login command in the commands of the rule.

The following command informs MAKE to process the MAKEFIN makefile using the first target in the makefile.

```
>MAKE MAKEFIN
```

If you want to require MAKE to rebuild everything in the makefile, ignoring timestamps, use the **ALL** option.

```
>MAKE MAKEFIN;ALL
```

You can specify any target as an entry point into the makefile. For example, if you wanted to rebuild only the reports of the FINANCE application, you would use the following command:

```
:MAKE MAKEFIN,FINRPTS
```

For more information on other **MAKE** options, refer to Chapter 1, "Commands", in the *LIBRARIAN/iX Reference Guide*.

The TOUCH Command

Because MAKE is driven by the MPE modification timestamp recorded in a file's label, it may be necessary to manipulate this timestamp directly. Along with MAKE, LIBRARIAN provides a command called **TOUCH** (available from the **File** menu) to make a file appear modified. In other words, the **TOUCH** command updates the MPE modification timestamp in the file label to reflect the current date and time.

If you touch a target file, it appears up-to-date. On the other hand, if you touch a dependency, it makes any target depending on it out-of-date. In this way you can selectively force or prevent MAKE from rebuilding a target.

Macros 9

This chapter describes how to create and use macros. Topics discussed in this chapter include:

- What are Macros?
- Sample Macro
- Filelists and Parameters
- Menus in Macros
- Conditional Expressions
- Looping in Macros
- The ALLOW Command
- Procedure Files
- AUTOXEQ Files
- Menus

Note

For information about executing macros and procedures, refer to the **XEQ** command in Chapter 1, "Commands", and Chapter 7, "Macro Control Language", in the *LIBRARIAN/iX Reference Guide*.

What Are Macros?

Macros are files that contain commands for LIBRARIAN to execute. You typically use macros to process a single file or a group of files. Macros can accept parameter values from a user. Macros can contain looping and conditional logic through the use of a special macro language.

You execute macros within LIBRARIAN by typing the name of a file containing LIBRARIAN commands, followed by an optional list of files and other parameters. Since macros are more flexible than steps (in fact, steps are frequently performed within macros), you can use macros to define operations too complex to be performed by a single step.

Some common uses of macros are:

- Create a single command that performs several LIBRARIAN steps and/or commands in sequence on a group of files.
- Perform a step several times against the same group of files, but with different destinations, such as to distribute a set of files to several systems.
- Perform a step or command with "hard coded" runtime parameters.
- Allow the user to execute commands which would normally require LIBRARIAN Manager or Application Manager capability.

The LIBRARIAN Manager can create macros in a secure location and make them available to all users. For MPE, this location is XEQ.OCSLIB, and, for UNIX, this location is /opt/ocs/ocslib/xeq. General users can, also, create macros for their own use. LIBRARIAN checks the current directory first, and then checks XEQ.OCSLIB (MPE) or /opt/ocs/ocslib/xeq (UNIX) for a macro when parsing commands.

Sample Macro

The following example shows a macro used to submit source for testing and to compile each program using the MAKE facility.

```
OPTION FILES=ABC-SUBMIT.ABC-MAINT.ABC,NOBREAK
ABC-SUBMIT !XEQLIST
MAKE ABCMAKE.PUB.ABCQA,%%[=P.OBJECT.ABCQA]
```

This simple macro uses the step ABC-SUBMIT to authorize files (**OPTION** statement), submits the files, and then recompiles each file with MAKE.

!XEQLIST is a list of the files authorized, created automatically by the macro processor. The **%%[ ]** parameter causes the MAKE statement to execute once for each file. The edit mask "=P.OBJECT.ABCQA" transforms the name of each source file into the corresponding object filename, which is the target name that is passed to MAKE.

Filelists and Parameters

Many macros accept a file reference like a step (as shown in the previous example), but this is not required. The following example uses the macro facility to execute **SHOWME**, followed by the **SHOWJOB** display of jobs currently executing.

```
SHOWME
SHOWJOB EXEC;JOB=@J
```

To require a file reference, use the **OPTION FILES** statement (as in the first example). If a step name is specified, the step definition is used to authorize the files; otherwise, the files are authorized in the same manner as for LIBRARIAN commands such as **XCOPY** or **XMOVE**.

Macros can contain a maximum of one hundred other parameters to be substituted at runtime. You can use these parameters for any string value, up to 80 characters.

You can set parameter values with the **PARM** statement and parameter references can appear anywhere in the body of the macro. They must appear in the format **%%*n***, where *n* is the parameter number (0 to 99).

The following example uses a parameter to request a project name from the user by presenting a menu of authorized projects (which LIBRARIAN provides in a file called PROJMENU), then checks in all the files associated with that project on an all-or-nothing basis.

```
MENU= PROJMENU
PARM 1;REQUIRED
ABC-CHECKIN.%%1;NOVIOLATIONS
```

Alternatively, the preceding example could have been coded to have the user specify the parameter on the command line, without presenting a menu.

```
ABC-CHECKIN.%%0;NOVIOLATIONS
```

Note that parameters are positional (the first parameter is 0, the second is 1, etc.)

The user performs this macro, called ABCIN, for project SR1234 by typing:

```
ABCIN SR1234
```

Menus in Macros

You can create your own menus as shown in the following example:

```
ECHO NULL
LMAINT
OUTPUT %SOURCE-FILES TO SRCFILES ;ALL
EXIT
ECHO STDLIST
PARM 3 ;MENU=SRCFILES ;TITLE=Source Menu ;PICKFILE
LOOP %%3
      ECHO %%*
NEXT
```

This example uses LMAINT to create an indirect list of files presented to the user as a menu. Selections are then displayed one per line. For more information, refer to the **PARM** command in Chapter 7 ,"Macro Control Language", in the *LIBRARIAN/iX Reference Guide*.

Conditional Expressions

Macros can include **IF/ELSE/ENDIF** conditional logic. Conditional expressions compare the values of strings, parameters, environment variables (UNIX), JCWs (MPE), and numbers, in addition to testing for the existence of files.

The following example checks for the existence of a text file by first applying an edit mask, and then checks out a source file if the text file does not exist.

```
OPTION FILES=OUT-SRC
LOOP
      IF EXISTS %%[=-T.TEXT.=]
         OUT-TEXT %%[=-T.TEXT.=]
      ELSE
         OUT-SRC %%[]
      ENDIF
NEXT
```

Looping in Macros

The macro control language supports the following looping structures:

- LOOP/NEXT
- REPEAT/UNTIL
- WHILE/ENDWHILE

The **LOOP/NEXT** structure works in either of two ways; it causes the execution of a block of commands for each:

- authorized file in the XEQLIST file, or
- record in a text file (fixed length record shorter than 80 characters).

The following example shows a macro which checks in COPYLIB members, then streams a job to update the master COPYLIB:

```
OPTION FILES=COPYLIB-IN
LOOP
      COPYLIB-IN %%[ ]
      STREAM
      !JOB COPYBLD,MGR.PROD
      !RUN COBEDIT.PUB.SYS
      LIB DCLIB.COPYLIB.PROD
      PURGE %%[=]
      COPY
      %%[=.COPYLIB.PROD]
      N
      %%[=]
      N
      EXIT
      !EOJ
      >
NEXT
```

COPYLIB member files (referred to by the %%[ ] variable) are authorized by the step (COPYLIB-IN) when executing this macro. Then, each file is moved to the production account, and a job is streamed to update the production COPYLIB. The equal sign (=) edit mask produces only the filename (without the group and account). The right angle bracket (>) is necessary to indicate the end of the stream.

Note 

The previous example is provided to demonstrate the use of **LOOP/NEXT** and the **STREAM** capability; a simpler solution to this problem is a macro that invokes MAKE, similar to the example earlier in this chapter.

The following shows a macro which distributes files to remote systems listed in a file called HOSTS.

```
LOOP HOSTS
      MFG DIST %MFG FILES TO %%*
NEXT
```

For each record in HOSTS, the files in the MFG FILES fileset are distributed by the step (MFG DIST) to the location defined by the contents of the HOSTS record (referred to by the %%* variable).

If the filename is absent, **LOOP/NEXT** works as in the first example, in which case you *must* include an **OPTION FILES** statement; otherwise, the **LOOP/NEXT** command(s) will have no files for which to loop.

Note 

Loops cannot be nested, but they can contain conditionals.

The **REPEAT/UNTIL** and **WHILE/ENDWHILE** structures cause the repetition of a block of commands until a conditional expression is true, or while a condition is true, respectively.

Nesting Macros

Nested looping is supported through nested OPTION FILES macros. LIBRARIAN keeps track of the nesting level, and opens a new XEQLIST1,2,3,...n as each nested macro is invoked. The following example checks in files specified by the user, and notifies the owner of every copy of each file checked in:

```
PROCEDURE ABC-IN
OPTION FILES=ABC-IN
LOOP
      SETJCW LIBOK=0
      CONTINUE
      ABC-IN %%[ ]
      IF LIBOK>0 THEN
      CONTINUE
      ABC-NOTIFY * AT @.@.@.@
      ENDIF
NEXT
END

PROCEDURE ABC-NOTIFY
OPTION FILES
LOOP
      MAIL %%(!OWNER), A new version of %% [ ] & has been
      checked in
NEXT
END
```

Reusing Macro Parameters

The **LOCALPARMS** parameter of the **OPTION** command allows macro parms to be independent of nested macros. Within nested macros, all parms are initialized to null values; original values are restored on return to the calling macro. This allows parms to be passed "by value", as arguments to the macro call.

The ALLOW Command

The **ALLOW** command temporarily allows you to perform functions that require user capabilities or step authorizations that general users do not possess. This is very useful, because it permits the LIBRARIAN Manager to grant users specific capabilities, limited to certain files and circumstances, without granting full capability. The following macro allows any user to orphan write mode files residing in your work group.

```
OPTION FILES=ABC-MYFILES,NOBREAK,NOHELP
ALLOW LIBMGR:GORP
SET !XEQLIST MODE=READ
ORPHAN !XEQLIST
ALLOW
```

In this example, a null step, ABC-MYFILES, has been created to perform the authorization by ownership. Selected users will be authorized on the Step Authorizations (SA) screen to perform this step (and hence, the macro), only for their own files.

The first instance of **ALLOW** provides the user with LIBRARIAN Manager capability to perform the restricted commands; the second instance of **ALLOW** restores the user's normal capabilities. The **NOBREAK** and **NOHELP** options are used so users cannot break while being allowed the capability, and so users cannot display the LIBMGR password.

Note that **ALLOW** is preferable in this situation to actually changing user identity with the **USER** command within the macro, as it preserves the original user ID in the audit trail.

It is recommended that you only use **ALLOW** in secure macro files.

Procedure Files

Procedure files are collections of macros in a single file, similar to a UDC catalog (MPE). The use of procedure files avoids the proliferation of macro files on disk, and allows to catalog multiple macros. Procedures in a procedure file begin with the **PROCEDURE** statement and end with the **END** statement, as in the following example:

```
PROCEDURE SJJ
SHOWJOB EXEC;JOB=@J
END
```

You can only invoke procedures if the procedure file has been loaded. For example:

```
>SET PROCEDURE TO ABCXEQS.XEQ.OCSLIB
```

Alternatively, you can load procedures by selecting the **Load Procedures** option from the **Macros** menu. Otherwise, procedure files and macros are identical.

AUTOXEQ Files

At startup, LIBRARIAN searches for a macro called AUTOXEQ.XEQ.OCSLIB (MPE) or /opt/ocs/ocslib/autoxeq (UNIX), and if found, performs it immediately. It then searches for a file called AUTOXEQ (MPE) or autoxeq (UNIX) in your current login directory and executes the file.

You can use this feature to set global parameters, or for each user to set a user ID and work environment. For example:

```
QUIET DISPLAY
USER FRED
SET PROCEDURE TO FREDXEQ
SET APPLICATION FIN
MENU OFF
```

In this example, you suppress LIBRARIAN informational messages and prompts, set your user ID, load a procedure file automatically, set the default application to FIN, and suppress menus so that you immediately go to the command line prompt.

Appendix A
Applications in Progress

There are special considerations when implementing LIBRARIAN for an application which is already undergoing modification.

- When you define the library for the application, make sure to identify the master files, as usual.
- Identify files that are being modified as secondary copies of the newly identified master files, even though they were not checked out with LIBRARIAN, by doing a checkout with the **INPROGRESS** parameter.

This appendix describes how you can implement LIBRARIAN with applications where work is already in progress. The following topics are discussed:

- Identifying secondary files
- Recording checkout

Identifying Secondary Files

The need to identify existing files as secondaries arises when you first implement LIBRARIAN for an application and files already exist in secondary locations; these secondary files need to be linked to their corresponding master files. Normally, you would copy files into those locations with a master-to-secondary step, but in this case you need to simulate the step without physically affecting the existing files in progress.

For example, assume you defined the library for the AP application and created the APOUT step to check out files from the AP account to the APDEVEL account. However, a programmer is currently modifying copies of two AP files: RCA.PUB.AP and RCB.PUB.AP, which were copied to APDEVEL before LIBRARIAN was installed.

Now you want to associate the files already in development with the newly defined master library without replacing the work that has already been done. The AP master files have serial access mode, and you want to protect these development copies by recognizing them as write-mode secondaries.

Recording Checkout

Use the **INPROGRESS** parameter with a defined master-to-secondary step, in this case APOUT, to record files as secondaries of specific master files. With the **INPROGRESS** parameter, LIBRARIAN performs all aspects of the step *except* physically copying the file. The file in the destination location is left as-is, but is tracked as a write mode secondary of a master file. Record the two files in the example above as secondaries in progress by typing:


```
>APOUT RCA.PUB.AP, RCB.PUB.AP ;INPROGRESS
```


```
>APOUT /ap/pub/rca, /ap/pub/rcb ;INPROGRESS
```

If you have work in progress in another secondary location, such as QA, which would normally be copied by a secondary-to-secondary step, you can record those files, as well, as write mode secondaries. To do so, create a temporary master-to-secondary step with the QA location as the destination and then using the **INPROGRESS** parameter on the step. After you use the step to record the files as secondaries in progress, delete the step.

LIBRARIAN/iX Glossary of Terms

Note  Terms that appear in *italics* in the following definitions have separate glossary entries.

A

Access Control

The attribute of a *master file* that determines how many *read/write mode* copies are allowed. The four access control levels are: *exclusive, read only, serial write, and multiwrite.*

Access Mode

The attribute of a *secondary file* that determines whether or not it can be checked in and replace its associated *master file.* A secondary in *write mode* can replace a master. A *read mode* can only replace a master through an *emergency checkin* that is configured to use the *PUSHREAD* parameter. A file's access mode is determined by *access control,* user request, *step* definition, and *default access mode* (precedence is in order listed).

Aging Policy

A *system profile* value that indicates how long log records are kept. When the *FLUSHLOG* utility is run, audit trail records that are older than the number of days specified in the aging policy are deleted.

Transactions associated with projects override this policy and are deleted only when the project status is flush pending.

Alternate prestep

A *prestep* that can be performed as an alternative to the defined prestep. Up to three alternatives can be defined for a *step.*

Annotate

Comments inserted by LIBRARIAN into source listings that indicate which lines were inserted/deleted for which *revision.* Date/time, related project and user who made the change are included.

Application

A site–defined organizational unit including a set of *master files* that are being controlled by LIBRARIAN, a set of *steps* for file movement/approval, and, optionally, a set of *projects* for tracking file changes associated with a particular work activity.

Application Manager

A *special user capability* assigned to the user responsible for the files and *steps* within an *application.*

Application fileset

The highest level *fileset* for an *application*.

Approval step

A *null step* that is required as a prerequisite for a subsequent step.

Authorization

The process of determining which files have been requested in a *transaction* and whether or not the rules permit the operation to be performed on each of these files. Authorization is based on the user who initiated the request and the current status of each file requested.

AUTOXEQ file

A *macro* that is executed before the first prompt/main menu appears. A file called AUTOXEQ that exists in the product account is executed prior to any AUTOXEQ file that might exist in the user's home directory.

Auto fileset descriptors

General locations that describe how *master files* are assigned automatically to *master filesets*. Descriptors can include or exclude files from filesets using *wildcards*. When you run *AUTOUPDATE*, introduce new files with a *pending master*, or perform a *checkin step* with the AUTOUPDATE parameter turned on, any previously *untracked files* in these locations get added to the appropriate master filesets.

Automatic Login ID

The login used when transactions require automatic logging in to a remote system.

Autoupdate

The process used to add *master files* to *master filesets* automatically based on predefined *auto fileset descriptors* that include or exclude files from filesets, typically using wildcards. *Pending masters* and masters not currently assigned to required filesets are added, typically during *checkin*, *new steps* and/or running of the AUTOUPDATE utility.

B

Baseline

The *master library* at a particular point in time. An *application manager* establishes a baseline by creating a *version*. This marks and protects all of the files in an application at that time, so that the application or any part of the application can be restored to that baseline any time in the future.

Base Revision

A *revision* that was current at the time a *baseline version* was created. The *version count (VCOUNT)* for a base revision is always zero and cannot be flushed until the version(s) of which it is a part is made *obsolete*.

Branch

A set of *revisions* that are made as a divergence from the main development path for a master file. A branch is created automatically when a previous revision is checked out. A branch can also be forced from the latest revision if the master is already checked out in *write mode*, or the user does not intend to check the file back in on the *trunk*. Whenever a new branch is created, a branch counter and *leaf* counter (both starting at 1) are appended as a pair to the original *revision ID*.

Branch revision

A *revision* that appears on a *branch*.

C

Checkin step

Any *step* which copies or moves a file from a *secondary location* into the *master library*, either retaining and replacing the existing master, introducing a new one or establishing a new branch .

Checkout step

Any *step* which copies a file from the *master library* into a *secondary location*, generally for modification by programmers.

Client

An MPE or UNIX implementation of LIBRARIAN where the LIBRARIAN data bases reside on a different system, but the user is able to perform all LIBRARIAN functions.

Command Mode

In command mode, the user enters LIBRARIAN commands at a command line prompt. Users can switch between command mode and *menu mode* by pressing the F2 function key.

Component filesets

Filesets that are subsets of higher–level filesets.

Composite prestep

A collection of *presteps* that must be performed before a subsequent step can be performed. Composite presteps also permit the specification of a date prerequisite.

D

Default access mode

The *access mode* that is assigned to a *secondary file* when neither the user or *step* explicitly specify the mode. The *access control level* for a file determines which access modes are allowed.

Delta file

A privileged (MPE) or hidden (UNIX) file that contains the history of changes made to an associated *master file*.

Deltas

A method for retaining and reconstructing previous revisions of *master files* that involves storing only the changes to files over time.

Dependency

A file that *make* evaluates with respect to some target to determine whether to invoke some action, such as a compile or link.

Destination

The target location when copying or moving a file.

Dummy target

A *make target* that does not correspond to an actual file. *Dependencies* of dummy targets are actual files that are always evaluated as targets themselves to determine whether they are out of date and need to be rebuilt.

E

Edit mask

A file expression that uses special editing characters to map one filename into another; e.g., source to destination name for a copy or move or *secondary* to *pending master name* for introduction of a new file.

Emergency checkin

A checkin that moves a *read mode secondary file* into the *library* with the *PUSHREAD* option. If a *write mode* copy exists, the *owner* is notified via a LIBRARIAN *mail* message, and an *exception* is recorded.

Exception Flag

An indicator that something special has happened related to a file such as an *emergency checkin, merge conflict* or previous *master revision* was restored at a time when the file was checked out. The exception flag must be cleared before any further operation on the file is allowed.

Exception message

A LIBRARIAN *mail* message that indicates that an exception flag has been placed on a file. This message is sent to the *owner* of the *write mode* copy of the file.

Exclusive access

The *access control level* that prevents *secondary* copies of a *master file* from being made.

Expiration date

The date when after which a file can be flushed using the *FLUSH* utility.

Expired file

A *read mode secondary* or *retained file* that is eligible to be flushed by the *FLUSH* utility.

Explosion

The creation of a list of files by expanding a *fileset, listfile,* or *wildcard* file specification for LIBRARIAN to *authorize.*

External

A file that resides on a system on which LIBRARIAN is not running, typically an unsupported platform, or system which is not on an accessible network. LIBRARIAN steps can be used to record movement to an external location, but cannot physically move the file or verify its existence. Users are responsible for transferring files (via tape or other means) for any transaction using the EXTERNAL option.

F

Fileset

A collection of files identified by a unique name assigned by the *Librarian Manager* (*master filesets*) or any user (*user filesets*). When requesting files, filesets can be referenced by preceding the fileset name with a percent sign (%). Because filesets contain collections of files that are related by some criteria other than physical location, and can span directories and systems, they are often referred to as *logical filesets.*

Note: In MPE, a fileset is any set of files that can be referred to using wildcards in name, group and/or account. LIBRARIAN refers to this as a *physical fileset.*

File structure (hierarchy)

The relationship of filesets, subsets and physical files within an application library.

Flush policy

The *system profile* policy that determines how many previous file *generations* to keep when the *FLUSH* maintenance utility is run.

FLUSHLOG

The maintenance utility that purges old log records that have aged beyond the *aging policy* specified in the *system profile.*

FLUSH

The maintenance utility that purges *expired files* and *obsolete versions.*

Flushed project

When a project is closed and then assigned a status of flush pending, log records associated with that project get flushed the next time the *FLUSHLOG* utility is run. After FLUSHLOG has been run, the project status is changed to flush, and the project can be deleted, if desired.

Flushed version

When a *version's* status has been changed to *obsolete*, base *revision* files that are a part of that version are flushed if they are not also part of a subsequent version. After *FLUSH* has been run, the version status is changed to flush, and the version can be deleted, if desired.

Flush pending

A *project status* that indicates that log records for the *project* should be purged when the *FLUSHLOG* utility is run.

FMAINT

The facility for creating and maintaining *user filesets*.

Forward versioning

An option on *checkout* to automatically search alternate *libraries* (usually previous versions) when a *master file* is not found in the expected *location* as defined by the checkout step. If the file is then found in an alternate location, it is brought forward as a *secondary* of a new *pending master* for the primary *application*.

G

Generation

Each time a file is checked in, a new generation is created. Previous generations of *master files* are stored in the *library* as *retained files* (usually compressed) or as *deltas*.

Generation count (GCOUNT)

A sequential number assigned to each *master file generation*. The current GCOUNT is the total number of times a master file has been replaced. When specifying GCOUNT as an option in a file request, a negative number indicates a generation relative to the latest generation.

Generic rule

A *target–dependency* relationship in *make* that uses *wildcards* (target) and edit masks (dependency) to determine what is out of date. Actual target and dependency names are substituted into the rebuild commands using *make macros*.

I

Indirect file

Also called a *listfile*, an indirect file is a text file that includes a list of filenames. This file can be used in *LIBRARIAN* commands as a convenient way of referencing files. Indirect files can be created in a text editor or through *LIBRARIAN's LMAINT* facility.

INPROGRESS

A parameter used with a *checkout step* that instructs LIBRARIAN to record the existence of a *write mode secondary* without physically copying the file from the *library*. This parameter is most often used when LIBRARIAN is initially implemented and some files are already being worked on or tested.

Intermediate revision

Master files that are retained between versions. The version count (VCOUNT) for intermediate revisions is always greater than 0.

L

Leaf Revision

Each *revision* on a *branch* is called a leaf, sequentially numbered from the start of the branch. Whenever a new branch is created, a branch counter and leaf counter (both starting at 1) are appended as a pair to the original *revision ID*.

LIBRARIAN

The program that controls and processes all file operations maintaining an audit trail of activity.

LIBRARIAN Manager

A *special user capability* assigned to the person responsible for configuring LIBRARIAN and defining site rules. The LIBRARIAN Manager has unrestricted access to all LIBRARIAN functions for all files.

Library

A library is the repository from which files are *checked out*, and to which they are subsequently *checked in*. Files are also distributed to production locations from the library. It is the 'official' collection of files that are under LIBRARIAN's control. Files in the library are called *master files*. The library provides a central point of control for changes to production source, object and data.

Listfiles

Also called an *indirect file*, a listfile is a text file that includes a list of filenames. This file can be used in *LIBRARIAN* commands as a convenient way of referencing files. Listfiles can be created in a text editor or through *LIBRARIAN's LMAINT* facility.

LMAINT

The facility for creating and maintaining *listfiles (indirect files)*.

Location

The group/account (MPE) or directory (UNIX) and *system* where a file exists or should be created.

Logical fileset

A meaningful name assigned to a collection of files not bound by physical boundaries. See *fileset*.

!LOGON, !LOGIN

A special wildcard that can be used in defining step source and destination *locations* to indicate that the user's login data should be substituted as appropriate. For MPE, this wildcard can be used for group, account and/or *system*. For UNIX, this wildcard is equivalent to '.' for current working directory and can also be used for *system*.

M

Macro

A set of *LIBRARIAN* and operating system commands for LIBRARIAN to execute. A macro *control language* provides programmatic control (conditions and loops) and parameter substitution. Parameter values can be system–defined or provided by the user via prompts and/or customized menus. Macros are analogous to MPE command files and UNIX scripts. Multiple macros can be combined in a single *procedure file*. Macros are also referred to as *XEQ files*.

Macro Control Language

The set of special commands and keywords that are used in macros to control flow of execution (IF...THEN...ELSE, REPEAT, WHILE, LOOP, GOTO) and allow for parameter substitution (tokens preceded by %%).

Mail

Mail includes messages that are sent from one LIBRARIAN user to another, or from *LIBRARIAN* notifying a user that an *exception* condition has occurred that affects that user's work.

Make

A utility that automatically rebuilds/recompiles components of an *application* when they change. Make reads a *makefile* that shows *dependencies* between application components and evaluates which components are out of date. Based on which components are out of date, make issues only the commands necessary to bring the application up to date.

Makefile

A text file that contains make rules. This file can have any name and can be created and maintained using any text editor. This file includes *target–dependency* relationships and commands required to bring each target up to date whenever their dependencies are changed. *Make macros* and *generic rules* can be used to reduce the size and complexity of a makefile.

Make macros

A shorthand that simplifies creating *makefiles*. Macro references are substituted with either user–defined or system–defined values when the

makefile is processed. For example, out–of–date *dependency* names can be substituted in generic command descriptions.

Master file

A file that is part of a defined *library* and reflects the most current production version.

Master fileset

A *fileset* defined by the LIBRARIAN Manager that includes *library* files.

Master library

The hierarchy of *master filesets* and associated *master files* for an *application*.

Memo

Text that provides documentation for a *transaction*. Memos are stored in the audit trail database and can be reviewed using *SHOWLOG*.

Menu Mode

The mode of *LIBRARIAN* operation in which users select LIBRARIAN functions from a set of pull–down menus. Users can switch to the command line prompt at any time by pressing the F2 function key.

Merge

An option available on *checkout steps* to combine source code changes from one or more *branches*. Conflicting changes are highlighted with comments in the source code, and should be resolved prior to the next step. Merge is only available if the *delta* feature is being used.

!MSUSER

A special *wildcard* that can be used in defining step destination *locations*. When the step is executed, the wildcard is replaced with the user ID of the user who originally checked out the file. For MPE, this wildcard can be used to fill in group or account. For UNIX, this wildcard can appear anywhere in the path name. This wildcard is typically used to reject files and move them from a test area back to the appropriate developer's work area.

Multi–write

The *access control* level that allows multiple *secondary files* with *write–mode* access.

N

New step

A *step* that introduces a previously *untracked file* to *LIBRARIAN* as a *secondary file*. The file is linked to a pre–existing *master file* or a *pending master* record is created. Rules governing introduction of new files on a step are configured on the PP (Pending Production Areas) screen.

Node

The actual device name associated with a system in a network. This name may or may not be the same as the LIBRARIAN *system ID*.

Null step

A *step* not involving any file movement. A null step is used to reflect some external action such as an approval. Null steps are used to control dependencies between steps; that is, they are used as *presteps*.

O

Obsolete version

When the *LIBRARIAN Manager* or *Application Manager* change the status of a *version* to obsolete, any *retained base revisions* associated with that version will be flushed the next time the *FLUSH* utility is run. Once a version is flushed, it can be deleted, if desired.

Operator

A *special capability* assigned to a user who can *flush* records in the log database and can restore previous revisions of files.

Orphan

Any file not currently being tracked by *LIBRARIAN* or a *master file* not associated with an *application*. Orphans can be created by a LIBRARIAN operation that causes a tracked file to become untracked (unknown to LIBRARIAN), or by operations that use the orphan option to create files in destinations that are not to be tracked.

!OWNER

A special *wildcard* that can be used in defining step destination *locations*. When the step is executed, the wildcard is replaced with the user ID of the user who currently owns the file. For MPE, this wildcard can be used to fill in group or account. For UNIX, this wildcard can appear anywhere in the path name. This wildcard is typically used to approve files in multiple developer work areas.

P

Parent Fileset

A *fileset* that includes *component* filesets.

Pending master file

A file that is being *tracked* as a *master library file*, but, because it is new, does not physically exist in the *library* yet. The associated *secondary* is called a *pending production file* and was introduced through a *new step* or through the use of LIBRARIAN's *forward versioning* feature.

Pending master mask

An *edit mask* used to automatically derive a *pending master file* name based on the name of the *secondary file* being introduced through a *new step*.

Pending production area

Any *location*(s) defined for a *step* where previously *untracked files* can be introduced as new *secondary files*. Steps with pending production areas are considered to be *new steps*.

Pending production file

A *secondary file* that was introduced using a *new step*. The *master file* does not currently exist in the *library*.

Permissions

A UNIX term used to indicate file access rights; a matrix of read, write, and execute access for owner, group and world.

Physical fileset

A collection of files that exist in a particular *location*. Physical fileset references include specific filenames, or names using standard operating system/shell *wildcards*.

Prestep

A *step* that must be completed successfully for a file before the next step in the *route* can be performed. Presteps are often *null approval steps*.

Procedure

A *macro* that is included in a file with other macros with a procedure header.

Procedure file

A file that contains multiple *macros*. Each macro has a procedure header indicating the name of the macro. Procedure files can be loaded and unloaded while using *LIBRARIAN*.

Project

A way of organizing *transactions* and associated files with a specific work activity.

Project fileset

A *user fileset* that is created automatically when defining a *project*. The fileset is maintained automatically when files are *checked out* or introduced as new files for the project. Files can also be added to this fileset in advance by a *Project Manager* using the *FMAINT* facility.

Project manager

A *special user capability* assigned to users who can create projects, modify project status and authorize users to work on projects.

Project menu

Whenever *projects* are associated with a particular *route*, users are asked to select the project that they are working on from a menu when checking files out or introducing new files.

Project status

A flag that determines what activities can be associated with a *project*.

PUSHREAD

A *step* option which allows a *read mode* copy to replace a *master file* or *write mode secondary* which has not been checked in yet. This option is typically used for *emergency steps*.

R

Read mode

The attribute of a *secondary file* that indicates it cannot replace the *master*. Read mode copies expire after a configured period of time and can be flushed using the *FLUSH* utility.

Read only

An *access control* level that only allows *read mode* copies of a file.

Read step

A *step* that copies a *master file* to a *secondary* location in *read mode*, with no intention for modification. An *expiration* policy can be applied, so that read mode copies created by the step can be cleaned up automatically with the *FLUSH* utility.

Receiver

A *system* that can receive files from other systems, but from which *LIBRARIAN transactions* cannot be initiated.

Release Step

Similar to a *read step*, a release step copies files from the *library* to a production *location* in *read mode*. Typically, these files do not expire, and the previous version is often *retained*.

Retained file

A previous *generation* of a file saved under a *LIBRARIAN*–generated name *"G########"*. Files are retained when the retain parameter is used on a *step* and the destination file is a *tracked master* or *secondary file*. *Base revisions* are always retained. If *deltas* are being used, changes to the previous generations are stored.

Revision

Any set of changes made to a *master file* through a *checkin* step. Revisions include all *generations* of a master file including the most current. *Leaves* and *branches* also make up the set of revisions for a file.

Revision ID

Revisions are identified by version name followed by a colon (:) followed by version count. If the revision is on a branch, branch and leaf count pairs are appended delimited with periods (.).

Route

A set of automated procedural controls for managing file changes and distribution. A route consists of a predefined file–movement path that reflects an established cycle. The route includes *steps* for all allowable movements of the files for that cycle.

Route Alias

When defining *projects*, a route alias can be defined to indicate that the project only applies to a particular *route*. The project name can be used in place of the route name when performing a step (i.e., step.project) to bypass the *project menu*.

Rule Administrator

Similar to the *LIBRARIAN Manager*, the Rule Administrator is a user with *special user capability* who can define *LIBRARIAN* rules such as steps and filesets, but is not automatically authorized to perform LIBRARIAN functions, and cannot create *user authorizations*.

S

Scan/Replace

A *LIBRARIAN* function that searches files for patterns of text, and optionally replaces the matches with user–defined text.

Scope

The attribute of a *step* that restricts which files the user can request. When copying or moving files, the scope specifies where files come from and where they can be copied. Steps can restrict by fileset, from location and to location.

Secondary file

Any copy of a *master file* or another secondary file. All secondaries are linked to a master (or *pending master*) either directly or indirectly, and are in *read or write mode*.

Secondary location

Any *location* where *secondary files* can be created.

Serial write

The *access control* level that allows only one *secondary file* at a time to have *write mode* access, preventing concurrent modifications.

Server

A system that has an implementation of *LIBRARIAN* which includes the LIBRARIAN databases. *Clients* access this database and other LIBRARIAN functions remotely.

Settings

LIBRARIAN session-level parameters that control the user's working environment.

Special user capability

See *user capabilities*.

Standard Rule

A *make* rule that associates specific *target*(s) with specific *dependencies*.

Step

A rule governing the copying and moving of files from one *location* to another. Steps are the basic building blocks of the *LIBRARIAN* file movement and control system. Steps are grouped into *routes* and are performed using system– and/or site–defined names.

Step parameter defaults

Options that control the behavior of a step, by default.

Step parameter overrides

If allowed, users can override *step parameter defaults* by specifying desired overrides.

Step refinements/exceptions

A *step* definition that includes rules for altering the destination *location* based on the from location, filecode (MPE), and/or *fileset* membership. The same criteria can be used to alter the type of movement (copy, move or null) or exclude files altogether from the step.

Step type

There are three types of steps: master–to–secondary (MS), secondary–to–secondary (SS) and secondary–to–master (SM). MS steps are steps that checkout or distribute files. SM steps are steps that check files in. SS steps encompass all steps in between, such as move to test and approvals.

System

A unique *node* within a network identified to *LIBRARIAN* with a unique *system ID*.

System ID

Used to *identify* systems to *LIBRARIAN* within a network. Optionally appears as a prefix to a filename delimited by ':' to indicate the appropriate system.

System Profile

A set of global parameters maintained by the *LIBRARIAN manager* that control how LIBRARIAN operates. Includes items such as flush policy, aging policy, date formats, etc.

T

Tag

A user–defined name for a particular *revision* of a file or files that can be used to identify them at a later time, even after they have been *retained.*

Target

Component of a make rule that is built from one or more dependencies using one or more commands. Object code and executables are examples of targets.

Tracked file

A file for which there is a record in the LIBRARIAN data base. Tracked files are *masters, secondaries* or *retained files* and movement operations are controlled by *LIBRARIAN* rules. All other files are *untracked files.*

Transaction

Any LIBRARIAN operation attempted either successfully or unsuccessfully on a set of files. Except for commands which provide information, all transactions are logged in the LIBRARIAN audit trail.

Trunk revision

A *revision* that is not checked in on a *branch.*

U

Untracked file

A file for which there is no record in the *LIBRARIAN* database. Ad hoc operations on these files conform to normal operating system security. Steps cannot be performed for untracked files.

User authorizations

The mechanism for determining who can do what. Authorizations can be defined for *steps* and *projects. Special user capabilities* can be assigned so that specific authorization is not required in some cases.

User capabilities

Grants users certain privileges that transcend standard *user authorizations*. These include *LIBRARIAN Manager, Application Manager, Project Manager, Operator, Rule Administrator* and *X capability*. If no special capability is assigned, authorization is required for steps, and other commands conform to normal operating system security.

User fileset

A *fileset* created and maintained by a user through the *FMAINT* user fileset module. User filesets allow users to group files for their convenience. Like *master filesets*, precede user filesets with % when referencing them in commands.

!USERID

A special *wildcard* that can be used in defining step source and destination *locations*. When the step is executed, the wildcard is replaced with the user ID of the user performing the step. For MPE, this wildcard can be used to fill in group or account. For UNIX, this wildcard can appear anywhere in the path name. This wildcard is typically used to check out file's into the developer's work area.

User ID

A unique identifier for a LIBRARIAN user that is password protected. Users are prompted for their User ID when initiating the *LIBRARIAN* program.

User password

Used to protect against unauthorized use of the *LIBRARIAN* system. Passwords are required and can be changed by the individual users.

V

Verify

The *LIBRARIAN* facility for reviewing file information on–line or off–line.

Version count (VCOUNT)

The sequential number that tracks the number of *generations* since the current *version* was defined.

Version

All the files in an *application*, as they were at a specific point in time.

Version ID

The name given to a version by a *LIBRARIAN* or *Application Manager*.

W

Wildcards

Special characters or tokens used in filenames to request multiple files that match a pattern, and/or to determine destination *locations*.

Work-in-progress

Untracked files that were in development and/or test prior to LIBRARIAN implementation. These files can be handled using the INPROGRESS parameter with a checkout step.

Write mode

The attribute of a *secondary file* indicating that it can replace its *master file* through an authorized *checkin step*.

X

XEQ file

A text file that contains the commands for a single *macro*. These macros are executed by filename.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X