Software Delivery

IBM

Standard Packaging Rules for z/OS-Based Products

SC23-3695-10

Software Delivery

IBM

Standard Packaging Rules for z/OS-Based Products

SC23-3695-10

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 165.

Eleventh Edition, June 2003

This book replaces the previous edition, SC23-3695-09, which is now obsolete.

If you have any comments regarding this book, address them to Packrule@us.ibm.com the owner of the Packaging Rules.

If you would like a reply, be sure to include your name, address, telephone number, e-mail address, or FAX number.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Significant changes or additions to text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to the following licensed programs:

- IBM SMP/E for z/OS and 0S/390 Version 3 program number 5655-G44
- z/OS Version 1, program number 5694-A01
- OS/390 Version 2, program number 5647-A01

Order IBM publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers' comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Owner, Standard Rules for Packaging z/OS-Based Products
Department FPLA, Mail Station P526
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1986, 2003. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book . . . vii
Who Should Use this Book . . . vii
Why Should You Follow the Rules? . . . vii
Important Terms . . . vii
Conventions for Rules, Restrictions, and Recommendations . . . viii
How This book is Organized . . . ix
Additional Information . . . ix

Summary of Changes . . . xi

Chapter 1. Introduction to z/OS Product Processes . . . 1
1.1 What Is Product Packaging? . . . 1
1.2 How Product Packaging and Product Processes Evolved . . . 1
1.2.1 Evolution of Product Packaging . . . 1
1.3 Tasks Included in the Product Processes . . . 2
1.3.1 Packaging and Distributing the Product . . . 2

Chapter 2. Assessing Your Product's Packaging Requirements and Considerations . . . 5

Chapter 3. Contents of the Product Package . . . 7
3.1 Relative File Tape . . . 7
3.1.1 Format and Contents of the RELFILE Tape . . . 7
3.2 Program Directory (Installation Manual) . . . 11
3.2.1 Contents of the Program Directory . . . 11

Chapter 4. SYSMOD Types and Relationships . . . 13
4.1 Types of SYSMODs . . . 13
4.1.1 Functions . . . 13
4.1.2 PTFs . . . 15
4.1.3 APAR Fixes . . . 16
4.1.4 USERMODs . . . 16
4.2 Defining SYSMOD Relationships . . . 17
4.2.1 Conditional and Unconditional Relationships . . . 18
4.2.2 Hierarchy of SYSMOD Types . . . 18
4.2.3 Specific SYSMOD Relationships . . . 19
4.2.4 Coexisting SYSMODs . . . 22

Chapter 5. Fundamental Packaging Considerations . . . 25
5.1 Installation Methods . . . 25
5.2 Evaluating SYSMOD Relationships . . . 27
5.3 Adding FMIDs . . . 28
5.4 Requirements for New Releases . . . 29
5.4.1 Consolidating Functions and Service for Elements . . . 30
5.5 Record Length, Record Format, and Block Size Requirements . . . 31
5.6 Specifying Copyright Information . . . 34
5.6.1 Program Directory (Installation Manual) . . . 34
5.6.2 Product Tape . . . 34
5.7 Specifying a Rework Date . . . 34

© Copyright IBM Corp. 1986, 2003

5.8 Shared Libraries . . . 35
5.9 Source Code . . . 35
5.10 Avoiding UCLIN . . . 36

Chapter 6. Elements and Load Modules . . . 37
6.1 General Packaging Rules, Restrictions, and Recommendations for Elements . . . 37
6.2 Element Ownership . . . 38
6.3 Using Aliases for Elements . . . 39
6.4 Data Element Types . . . 39
6.4.1 USERx Data Types . . . 41
6.5 Shared Load Modules . . . 41
6.6 Generation Macros . . . 42
6.7 Sample JCL and Data . . . 43
6.8 Language-Sensitive Elements . . . 47

Chapter 7. Using MCS to Define Products . . . 49
7.1 ++FUNCTION Statement . . . 49
7.1.1 Specifying the SYSMOD ID (sysmod_id) . . . 50
7.1.2 Identifying the REWORK Date (REWORK) . . . 50
7.1.3 Specifying the Prefix for RELFILE Data Sets (RFDSNPFX) . . . 50
7.1.4 Specifying Copyright Information . . . 53
7.2 ++VER Statement . . . 53
7.2.1 General Packaging Rules (++VER) . . . 54
7.2.2 Identifying the SREL . . . 54
7.2.3 Identifying a SYSMOD's Base Function (FMID) . . . 54
7.2.4 Deleting SYSMODs (DELETE) . . . 55
7.2.5 Specifying Mutually Exclusive SYSMODs (NPRE) . . . 58
7.2.6 Specifying Prerequisite Relationships (PRE) . . . 58
7.2.7 Superseding SYSMODS (SUP) . . . 59
7.2.8 Defining Ownership (VERSION) . . . 62
7.3 ++IF Statement . . . 64
7.3.1 Specifying the Function to which the Condition Applies (FMID) . . . 64
7.3.2 Specifying Requisite Conditions (REQ) . . . 65
7.4 ++element Statement . . . 66

Chapter 8. Using MCS to Manipulate Elements and Load Modules . . . 71
8.1 Moving Elements and Load Modules (++MOVE) . . . 72
8.2 Renaming Load Modules (++RENAME) . . . 74
8.3 Deleting Load Modules (++DELETE) . . . 75
8.4 Deleting Elements from Libraries and SMP/E Data Sets . . . 77
8.5 Enabling Load Module Changes at the CSECT Level (++MOD CSECT) . . . 77
8.6 Defining Ownership of Elements (++element VERSION) . . . 78

Chapter 9. Using JCLIN . . . 81
9.1 Providing JCLIN Data for Function SYSMODs . . . 81
9.2 When Do You Need JCLIN? . . . 82
9.3 General Packaging Rules for JCLIN Data . . . 83
9.4 Assembler Steps . . . 84
9.5 Copy Steps . . . 84
9.5.1 Considerations for the SELECT Statement for Copy Operations . . . 86
9.6 Link-Edit Steps . . . 87
9.6.1 JCLIN Processing of DD Statements in Link-Edit Steps . . . 90

9.6.2 Link-Edit Control Statements 90
9.6.3 Link-Edit Attribute Parameters 99
9.6.4 Cross-Product Load Modules for Products Installed in the Same Zone 100
9.6.5 Cross-Product Load Modules for Products Installed in Different Zones 102
9.6.6 Adding or Changing Load Modules in a PTF 103
9.7 Examples of JCLIN Data 104
9.7.1 JCLIN Data for Modules 104
9.7.2 JCLIN Data for Macros and Source 108
9.7.3 JCLIN Data for an Assembler Step to Create a Module from Source . 109
9.7.4 JCLIN for Using the Link-Edit Automatic Library Call Function 109
9.7.5 JCLIN Data for Load Modules Residing in a Hierarchical File System or Java Archive file 112

Chapter 10. Naming Conventions 115
10.1 Component Identifier (COMP ID) 115
10.2 SYSMOD IDs 115
10.3 Element, Alias, and Load Module Names 115
10.3.1 NLS Considerations for Element Types 117
10.3.2 Elements with the Same Name 117
10.3.3 Alias Names 117
10.4 Library Names 117

Chapter 11. Packaging for National Language Support (NLS) 121
11.1 Element Types for Translated Data Elements 122
11.2 Planning the Physical Media for NLV 123

Chapter 12. Packaging for Special Situations 125
12.1 High-Level Languages 125
12.1.1 Support in SMP/E Release 7 and Later for the Automatic Library Call Facility 125
12.1.2 If You Cannot Use the Automatic Library Call Facility 125
12.2 Using the C Language Prelinker 128
12.2.1 Example of a Product Requiring the C Prelinker 129
12.3 Packaging Workstation Code to Be Installed on the Host 130

Chapter 13. SYSMOD Packaging Examples 135
13.1 Conventions Used in This Chapter 135
13.2 Example 1: A Stand-Alone Function 136
13.2.1 Initial Release 136
13.2.2 PTF Service for the Initial Release 137
13.2.3 PTF Service That Depends on Previous Service 137
13.2.4 Ensuring That a Fix for a Previous Release Is Not Lost 138
13.2.5 Replacing the Initial Release 139
13.3 Example 2: Corequisite Base Functions 141
13.3.1 Initial Releases of Corequisite Functions 141
13.3.2 PTF Service for One of the Base Functions 142
13.3.3 Cross-Product Service between Corequisite Base Functions 142
13.3.4 Deleting and Superseding a Base Function 143
13.4 Example 3: Dependent Functions 144
13.4.1 Initial Release of a Dependent Function 144
13.4.2 PTF Service for a Dependent Function 145

13.4.3 Corequisite PTFs with an Element Common to the Base and Dependent Functions 145
13.4.4 Corequisite PTFs with All Elements Common to Base and Dependent Functions 148
13.4.5 Deleting a Dependent Function Without Superseding It 151
13.4.6 Establishing the Order of Additional Dependent Functions 151
13.4.7 Conditional Corequisite Dependent Functions 152
13.5 Example 4: Base Functions with Prerequisites 152
13.5.1 Initial Release of a Base Function with a Functional Prerequisite . . 152
13.5.2 Dependency on an SPE or Service for Another Base Function . . . 153
13.5.3 Cross-Product Service for a Base Function with a Prerequisite . . . 154
13.6 Example 5: Mutually Exclusive Dependent Functions 155
13.7 Example 6: Functions Supporting More Than One Language 156
13.7.1 A Base Function Supporting Two Languages 156
13.7.2 PTF Service for Language-Sensitive Elements 157
13.7.3 Supporting Two Languages for a Base Function and Its Related Dependent Function 158
13.7.4 PTF Service for Common Language-Sensitive Elements 159
13.8 Changing the Contents of Products 160
13.8.1 Adding Elements 161
13.8.2 Combining Elements 161
13.8.3 Migrating Elements by Updating Both Functions 162
13.8.4 Migrating Elements by Using a PTF 163

Notices 165
Trademarks 165

Glossary 167

Publications and Classes 177
Related Publications 177
Classes and Self-Study Courses for SMP/E Product Packaging 177

Index 179

About This Book

This book refers to the IBM SMP/E V3R1 and z/OS manuals using generic titles. See “Publications and Classes” on page 177 to map to the appropriate reference manual for your system.

This book is designed to provide product developers with rules, requirements, and recommendations regarding how to package *z/OS installable* products. (A *z/OS-installable* product is any product that is installed from any medium directly onto an z/OS system, regardless of the system on which the code is intended to run.)

Who Should Use this Book

This book is intended for anyone responsible for the planning or packaging of a product that is z/OS installable.

Before using this publication, you should be familiar with the *SMP/E User's Guide*, to acquire a general understanding of SMP/E and how to use it to install SYSMODs. You should also be familiar with the *SMP/E Commands* and *SMP/E Reference* manuals, which contain more detail about SMP/E syntax.

Why Should You Follow the Rules?

The rules are intended to benefit both customers and internal development, installation, and service processes.

By following the rules, recommendations, and restrictions in this book, you can:

- Improve customer acceptance of your products
- Make it easier for customers to install and maintain your product on an IBM system
- Reduce the time and effort spent to analyze packaging problems, rework product packages, and test the associated deliverables
- Ensure that your product can be processed by common installation and distribution vehicles for z/OS based products

Important Terms

Throughout this book, *SMP/E* refers to SMP/E Release 5 or later. Specific references to a higher level of SMP/E are noted as appropriate.

The following terms are **highlighted** throughout this book as shorthand for the various processes involved in packaging. (The name of a process generally corresponds to the group or organization that performs that process. For example, the development process is carried out by the development organization.)

Process **Supporting Organizations**

Development product development, including product owners, designers, planners, programmers, and build groups

© Copyright IBM Corp. 1986, 2003

Service Software Service Business Development, Software Service Delivery planners, Service Process Management, Service Teams,

Distribution Software and publication centers worldwide

Installation Customer location whose system programmers install and maintain products and service

A glossary is provided at the end of this book to define additional terms that may not be familiar to you.

Conventions for Rules, Restrictions, and Recommendations

Rules, restrictions, and recommendations for the various packaging processes follow particular conventions.

For rules:

- The text of rules is enclosed in a box.
- "Must" is used instead of "should".

Following is an example of how a rule would be indicated:

Packaging Rule
☐ *n.* All elements must....

Restrictions are based on the limitations of SMP/E or IBM processes.

Recommendations indicate a preferred method of handling a situation; however, there may be other acceptable alternatives.

Restrictions and recommendations are indicated with text indicating the beginning and end of a small section of recommendations or restrictions; for example:

IBM Restriction

The **IBM** process does not support ...

End of IBM Restriction

Packaging Recommendation

It is recommended that all products ...

End of Packaging Recommendation

Also, note that ++*element* is the generic nomenclature for the MCS that identifies elements.

How This book is Organized

Table 1 describes the way this book is organized and what each chapter contains.

Table 1. Organization of This Book

Chapter	Description
Chapter 1, "Introduction to z/OS Product Processes"	Provides an introduction for developers who are relatively new to the MVS product processes, or who need only general information.
Chapter 2, "Assessing Your Product's Packaging Requirements and Considerations"	Lists questions that you can answer to help plan your packaging requirements.
Chapter 3, "Contents of the Product Package"	Describes the contents of the product package.
Chapter 4, "SYSMOD Types and Relationships"	Introduces you to SMP/E packaging concepts.
Chapter 5, "Fundamental Packaging Considerations"	Describes some fundamental topics for software packaging.
Chapter 6, "Elements and Load Modules"	Describes considerations for packaging the elements that make up a product.
Chapter 7, "Using MCS to Define Products"	Describes how to use MCS to package a product.
Chapter 8, "Using MCS to Manipulate Elements and Load Modules"	Describes how to perform basic operations on elements.
Chapter 9, "Using JCLIN"	Describes when to use JCLIN and how to use JCLIN.
Chapter 10, "Naming Conventions"	Describes the naming conventions for components, elements, libraries, and SYSMOD IDs.
Chapter 11, "Packaging for National Language Support (NLS)"	Provides information about NLS packaging considerations.
Chapter 12, "Packaging for Special Situations"	Provides information about some packaging areas that require special handling.
Chapter 13, "SYSMOD Packaging Examples"	Provides examples of SYSMOD packaging.
"Glossary"	Describes the terms used in this book.
"Publications and Classes"	Lists books and classes that provide additional useful information.

Additional Information

To understand and use much of this book, you need an understanding of SMP/E appropriate to your responsibilities. See "Publications and Classes" on page 177 for a list of SMP/E books that you may find helpful.

Summary of Changes

This book has been completely rewritten.

Chapter 1. Introduction to z/OS Product Processes

This chapter explains the following:

- What is meant by product packaging
- How packaging and processes for z/OS products evolved
- Tasks included in the product processes
- What each of the product processes does

1.1 What Is Product Packaging?

A program consists of elements such as modules, macros, and other types of data. Packaging is the science of building these elements into a deliverable product that can be installed and maintained on a computer system.

For z/OS systems, *SMP/E* is used to install a product, install changes (service, user modifications, new functions) to the product, and track the current status of each of the elements of the product. All products and service for z/OS installable products must be packaged so that they can be installed and maintained by SMP/E.

For SMP/E to install a product and service for that product, you must code SMP/E *modification control statements (MCS)* for the elements. MCS describe the elements of the product and any relationships the product has with other products that may also be installed on the same z/OS system. The combination of elements and MCS statements is called a *system modification (SYSMOD)*.

Product packaging includes combining the appropriate MCS statements with the elements of a program to create one or more SYSMODs, then putting the SYSMODs in the proper format on a *relative file tape (RELFILE tape)*. This relative file tape is used to distribute the product to customers.

1.2 How Product Packaging and Product Processes Evolved

This section describes how product packaging has evolved, and how specialized processes for z/OS products have changed.

1.2.1 Evolution of Product Packaging

The way in which z/OS systems have provided products has changed through the years. From the total system replacements of the early days, packaging has evolved to individual products, and to custom-built packages of multiple products.

- Individual Products - To make the software even more independent from the hardware and to allow a broader scope of independent software development, MVS release 3.8 restructured the software into many discrete functional areas identified by one or more function modification identifiers (FMIDs).
- SMP4 - MVS 3.8 included SMP Release 4 (SMP4), which supported function, PTF, APAR, and USERMOD SYSMODs. With SMP4, each functional area became separately installable and could be developed on asynchronous schedules (with proper considerations for dependencies on other functional areas). These functional areas are internally referred to as "products."

© Copyright IBM Corp. 1986, 2003

- SMP/E - To further enhance SYSMOD processing, SMP/E was developed, which uses zones in a VSAM data set (the SMPCSI) to manage the system's target and distribution libraries. These zones can be defined by the user to manage the increasingly complex relationships between products.

  To install an individual product, the customer uses SMP/E to install function SYSMODs, which contain the software, install logic, and JCLIN data for the product. For some products, the customer must also do a system, subsystem, or product generation to provide some job streams and SMP/E JCLIN data. SMP/E is also used to install preventive service and corrective service, with improved handling of exception SYSMODs.

1.3 Tasks Included in the Product Processes

The product processes carry out these general tasks:

- Planning for the product
- Designing and developing the product
- Packaging and distributing the product
- Packaging and distributing service
- Packaging and distributing service updates
- Ordering and installing the software

1.3.1 Packaging and Distributing the Product

Figure 1 on page 3 is an overview of the typical steps **development** must follow to get elements and JCLIN data from the development libraries into the SYSMOD format that will be used to distribute a product and its related service. Once the elements for the product have been coded, these are the basic steps to follow to package those elements:

1 **Integrate the code:** Use the available tools and procedures to create a set of data sets that SMP/E can process.

Create any required JCLIN data.

Collect the elements and JCLIN data from the various development libraries.

Note: If you plan to distribute updates for macros or source later on, make sure they are initially shipped with sequence numbers in columns 73-80. Otherwise, SMP/E is not able to install the updates.

Compile and assemble the macros and source to create the object modules.

Link-edit the resulting object modules. The format of the link-edited modules must be the same as the format produced by the MVS/370, MVS/XA, or MVS/ESA linkage editor.

2 **Build the SYSMOD package:** Use the available tools and procedures to create files and a relative file tape that SMP/E can process.

a. Create a sequential data set that contains the MCS for the elements.

b. Create the relative files by unloading the integrated data sets. The format of the unloaded data sets must be the same as the format produced by the IEBCOPY utility.

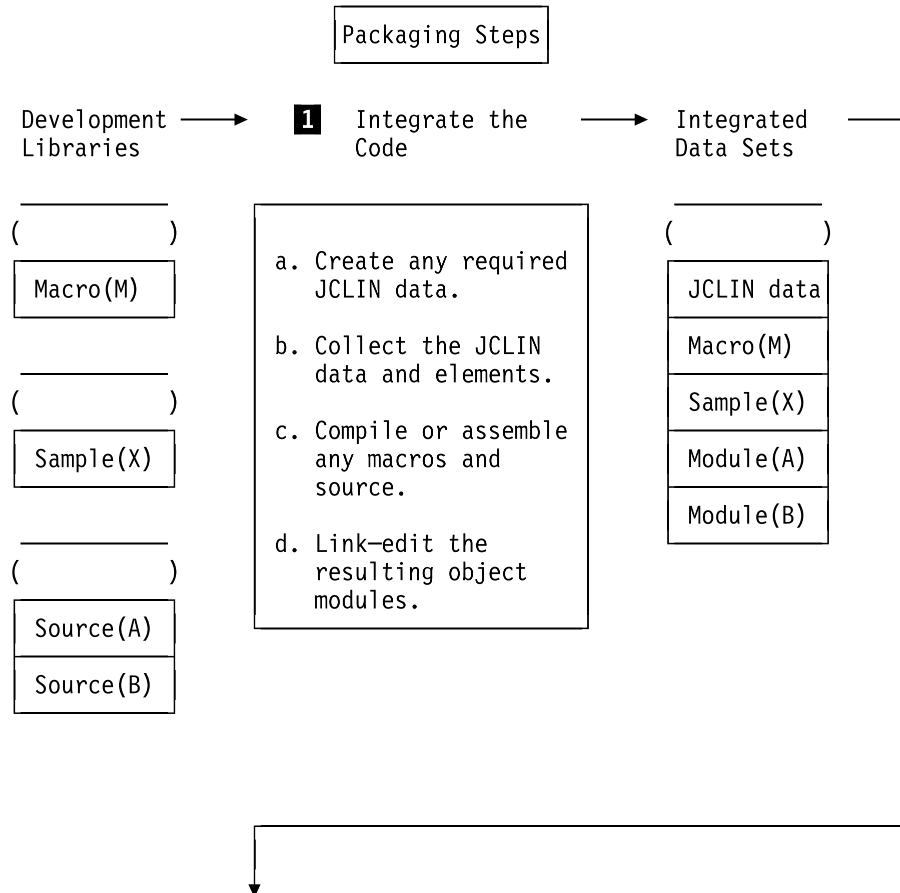


Figure 1 (Part 1 of 2). Getting Code from Development Libraries into SYSMOD Format

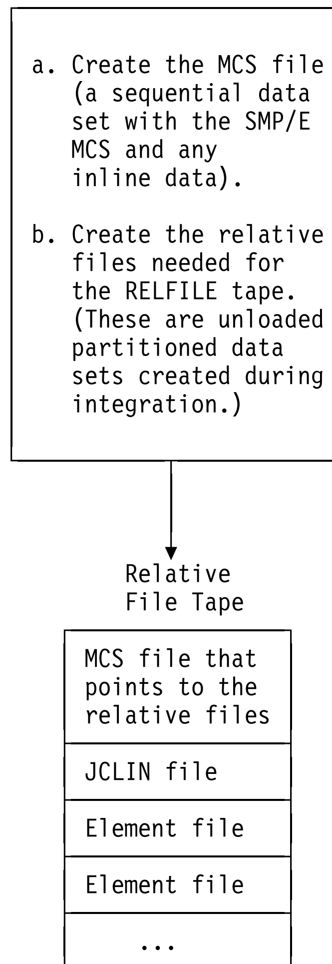


Figure 1 (Part 2 of 2). Getting Code from Development Libraries into SYSMOD Format

Working with a publication printer, **development** produces the materials for the product package. **Development** then sends these materials to **distribution** and **service** for processing. **Service** and **distribution** load the product package into their databases, validate the package, and distribute the package in its final form. If **distribution** or **service** discovers any problems with the product package, its staff works with **development** to resolve the problems. **Development** keeps **service** and **distribution** informed of the impacts of these problems.

Chapter 2. Assessing Your Product's Packaging Requirements and Considerations

The following questions help you determine:

- Your packaging requirements
- Areas you need to address
- Where to find the information you need

__ 1. Does the product use the standard SMP/E installation path (RECEIVE, APPLY, ACCEPT)?

If so, see 9.3, “General Packaging Rules for JCLIN Data” on page 83

__ 2. Is the product a base or dependent function?

See 4.1.1, “Functions” on page 13

__ 3. What SRELs does the product install in?

See 7.2.2, “Identifying the SREL” on page 54

__ 4. Do you require language support?

See the following:

- Chapter 11, “Packaging for National Language Support (NLS)” on page 121
- 13.4, “Example 3: Dependent Functions” on page 144

__ 5. Is this a new version, release, modification, or replacement?

See the following:

- 4.2.3.4, “Deleting and Superseding SYSMODs” on page 20
- 5.2, “Evaluating SYSMOD Relationships” on page 27

__ 6. Do you use high-level languages?

See 12.1, “High-Level Languages” on page 125

__ 7. Does your product use modules from another product?

See the following:

- 4.2.3.2, “Corequisite SYSMODs” on page 20
- 6.5, “Shared Load Modules” on page 41
- 8.5, “Enabling Load Module Changes at the CSECT Level (++MOD CSECT)” on page 77
- Chapter 12, “Packaging for Special Situations” on page 125
- 13.3.3, “Cross-Product Service between Corequisite Base Functions” on page 142
- 13.5.3, “Cross-Product Service for a Base Function with a Prerequisite” on page 154
- 13.8, “Changing the Contents of Products” on page 160

© Copyright IBM Corp. 1986, 2003

__ 8. Are there dependencies on other products, either within the same zone or residing in different zones?

See the following:

- 5.8, “Shared Libraries” on page 35
- 13.3.3, “Cross-Product Service between Corequisite Base Functions” on page 142
- 13.5.3, “Cross-Product Service for a Base Function with a Prerequisite” on page 154

Chapter 3. Contents of the Product Package

The product package contains these items:

- Relative file (RELFILE) tape
  - Object code only (OCO)
  - Source code (optional)
- Program directory (installation manual)

This chapter describes the packaging rules for these items and points to sources of requirements for submitting software publications.

3.1 Relative File Tape

A *relative file* tape, or *RELFILE* tape, is a standard label tape made up of two or more files. It contains a file of the MCS for one or more functions, and one or more relative files containing unloaded source data sets and unloaded, link-edited data sets containing executable modules. The relative files may also contain other data, such as sample procedures. These unloaded partitioned data sets must be in a format that can be installed on an z/OS system or subsystem by SMP/E.

Note: You can create a RELFILE tape as either an actual tape or as a tape image. When you see references to "RELFILE tape", they apply to tape images as well as to tapes.

Also, bear in mind that creating a RELFILE tape does not force users to receive the SYSMOD from tape. For example, a user may choose to load the RELFILE tape to DASD data sets, and then receive the SYSMODs from DASD. Or, a user may be sent a SYSMOD electronically, read the files into DASD data sets, and receive the SYSMOD from DASD.

The following section discusses the format and contents of the RELFILE tape.

3.1.1 Format and Contents of the RELFILE Tape

Table 2 on page 8 is an example of a RELFILE tape for two function SYSMODs: HBBZ100 (a base function) and JBBZ1B0 (a dependent function for the base function).

Note: The format and content of the copyright statement are dictated by law, not by this book. The example used is only an example, and may not be appropriate for any particular product. Consult your Intellectual Property Law department for the correct statement for your product.

Table 2. Example of a RELFILE Tape

File	Data Set Name	Contents
1	SMPMCS	++FUNCTION(HBBZ100) REWORK(2003160) FILES(3) RFDSNPFX(IBM) /**/ /* THIS PRODUCT CONTAINS "RESTRICTED MATERIALS OF IBM" */ /* 5665-123 (c) COPYRIGHT IBM CORP. 1990, 1994 */ /* ALL RIGHTS RESERVED. */ /* US GOVERNMENT USERS RESTRICTED RIGHTS */ /* - USE, DUPLICATION OR DISCLOSURE */ /* RESTRICTED BY GSA ADP SCHEDULE CONTRACT */ /* WITH IBM CORP. */ /* LICENSED MATERIALS - PROPERTY OF IBM */ /**/ . ++VER(Z038). ++JCLIN RELFILE(1). ++MOD(A) DISTLIB(GIMMODS) RELFILE(2). ++MOD(B) DISTLIB(GIMMODS) RELFILE(2). ++MAC(X) DISTLIB(GIMMACS) RELFILE(3). ++FUNCTION(JBBZ1B0) REWORK(2003160) FILES(3) RFDSNPFX(IBM) /**/ /* THIS PRODUCT CONTAINS "RESTRICTED MATERIALS OF IBM" */ /* 5665-123 (c) COPYRIGHT IBM CORP. 1990, 1994 */ /* ALL RIGHTS RESERVED. */ /* US GOVERNMENT USERS RESTRICTED RIGHTS */ /* - USE, DUPLICATION OR DISCLOSURE */ /* RESTRICTED BY GSA ADP SCHEDULE CONTRACT */ /* WITH IBM CORP. */ /* LICENSED MATERIALS - PROPERTY OF IBM */ /**/ . ++VER(Z038) FMID(HBBZ100). ++JCLIN RELFILE(1). ++MOD(A) DISTLIB(GIMMODS) RELFILE(2). ++MOD(C) DISTLIB(GIMMODS) RELFILE(2). ++MAC(Y) DISTLIB(GIMMACS) RELFILE(3).
2	IBM.HBBZ100.F1	Unloaded partitioned data set containing member HBBZ100, which is JCLIN data for function HBBZ100
3	IBM.HBBZ100.F2	Unloaded partitioned data set containing modules A and B for function HBBZ100
4	IBM.HBBZ100.F3	Unloaded partitioned data set containing macro X for function HBBZ100
5	IBM.JBBZ1B0.F1	Unloaded partitioned data set containing member JBBZ1B0, which is JCLIN data for function JBBZ1B0
6	IBM.JBBZ1B0.F2	Unloaded partitioned data set containing modules A and C for function JBBZ1B0
7	IBM.JBBZ1B0.F3	Unloaded partitioned data set containing macro Y for function JBBZ1B0

Packaging Rules (RELFILE Tape: Format and Contents)

- ☐ Rule 1.1. All product tapes must be RELFILE tapes.
- ☐ Rule 1.2. All files on a z/OS installable product tape must be SMP/E-installable.

Packaging Rules (RELFILE Tape: Format and Contents)

- ☐ Rule 2. The files must be in this order:
  - – The SMPMCS file (only one)

    Notes:

    1. If the package consists of several tapes, there must be one SMPMCS file per logical RELFILE tape.
    2. If the package consists of a base function and related dependent functions, the MCS for a base function must precede those for all its related dependent functions.

  - – The relative files

    Note: If the tape contains more than one function, the relative files must be in this order:

    1. The relative files for the first SYSMOD defined in the SMPMCS file. The order of each of these files must correspond to the value of the RELFILE operand specified on ++JCLIN and element statements.
    2. The relative files for the second SYSMOD defined in the SMPMCS file, and so on.

- ☐ Rule 3. The SMPMCS file must be a sequential data set consisting of 80-byte, fixed-length records.
- ☐ Rule 4. All the other files on the tape or set of tapes must be relative files for the functions defined in the SMPMCS file.
- ☐ Rule 4.1. The data set name of each relative file must be *hlq.fmid.***F***nnn*, where:

  hlq is the high-level qualifier of your company. This prefix should be used by all products.

  fmid is the FMID of the function to which the file is related.

  F*nnn* is the letter "F" followed by the number specified on the RELFILE operand of the corresponding MCS in the SYSMOD, up to a maximum of 999. Do not use leading zeroes in the RELFILE number.

  Note: The high-level qualifier *hlq* can be used only if RFDSNPFX is specified on the ++FUNCTION statement.

- ☐ Rule 5. All the elements for a function SYSMOD must be on the same logical tape as the SMPMCS file that defines the function.
- ☐ Rule 6. There can be only one element with the same name in a given relative file. This includes element names and element alias names.
- ☐ Rule 7. Each relative file must contain partitioned data sets that were unloaded in IEBCOPY format.

Packaging Rules (RELFILE Tape: Format and Contents)

- ☐ Rule 8. Sequential data sets must be packaged as members of a partitioned data set so that they can be unloaded by IEBCOPY into a relative file. If the product does not require OS/390 Release 7 or later, a postinstallation job can be provided to copy such an element into a sequential data set.
- ☐ Rule 9. Modules must be in link-edited format. (This is RECFM=U, undefined record format.) The input parameters used for the link-edited format must include NCAL. Providing modules in link-edited format eliminates the need for the LEPARM operand and other data that is required on the ++MOD statement when modules are provided inline. Contrast with restriction 16.1 in 9.3, "General Packaging Rules for JCLIN Data" on page 83 regarding what to do for a PTF that introduces a new ++MOD requiring link-edit parameters other than the default.
- ☐ Rule 9.1. RECFM=U RELFILEs must be blocked at 6144, so that they can be reblocked upwards at install time.

  Note: See Rule 33.1 for block size requirements for target and distribution libraries.
- ☐ Rule 9.2. ++MOD elements must not contain linkage editor control statements other than IDENTIFY and SETSSI inline.

  Note: This rule is applicable only to PTFs.
- ☐ Rule 10. VSAM data set elements must be in AMS REPRO format.
- ☐ Rule 11. The partitioned data sets to be unloaded must have a member for each element MCS, plus a directory entry for each ALIAS associated with an element MCS. Likewise, each member in a RELFILE must be defined by an element MCS.
- ☐ Rule 13. If a member in a relative file contains JCLIN data for a SYSMOD, the member name must match the function's FMID.

Packaging Rules (RELFILE Tape: Volume Serial Numbers)

- ☐ Rule 13.2. If two tapes have the same volume serial number (VOLSER), they must contain the same FMIDs. It is permissible for different SUP levels of the same FMIDs to use the same VOLSER.

SMP/E assumes that modules on RELFILE tapes are link-edited and were unloaded in IEBCOPY format. SMP/E invokes the IEBCOPY utility, not the linkage editor, when copying LMODs. The IEBCOPY utility requires that all partitioned data sets have the same format.

Modules should be single-CSECT load modules.

3.2 Program Directory (Installation Manual)

The program directory (installation manual) is a document shipped with each release of a product. Its primary purpose is to document the installation of the product.

The program directory is part of the informal documentation of the product. It is not a vehicle for changes not related to installation or for updates that are better handled in a technical newsletter or replacement publication.

3.2.1 Contents of the Program Directory

The program directory model explains what information is to be included in the program directory. The program directory performs the following functions:

- Describes all the machine-readable material and publications
- Documents which systems, concurrent programs, and machines are required
- Provides details on how to install the product.
- Documents the support that is available for the product.
- Identifies program and service levels when communicating with personnel.
- Identifies the resources needed to install the program, and the impact of its use on an existing data processing system.

Packaging Rules (Program Directory: Contents)

- □ Rule 17. If the program directory is used worldwide, use generic terms. For example, use the term “Software Distribution.”
- □ Rule 18. The installation instructions must describe the installation method used and the step-by-step procedures for installing the product. This section must also describe the procedures to activate the product function, unless this information is contained in another formal publication.

  Note: Installation in this context means those instructions that include the necessary information to complete a RECEIVE-APPLY-ACCEPT or RECEIVE-ACCEPT BYPASS(APPLYCHECK)-GENERATE.
- □ Rule 18.1. Every product must have a Program Directory unique to itself.

Packaging Rules (Program Directory: Contents)

- ☐ Rule 18.2. The following applies to composite products ('suite', 'server', etc.) that include existing products as components:
  - If the individual products are no longer available, the composite product's unique Program Directory must completely document the installation of these components and the composite product must not ship the individual Program Directories of the components.
  - If the individual products are still available, the composite product's unique Program Directory should completely document the installation of these components, but the composite product may ship the individual Program Directories of the components if all of the following are true:
    - The unique Program Directory refers to these Program Directories as necessary.
    - The individual Program Directories do not contain any documentation that does not apply to the composite product (for example, documentation about an FMID that is not shipped in the composite product).
- ☐ Rule 19.2. If the program directory includes any information reproduced from the product tape (SMPMCS, JCLIN, jobs, etc.), this information must exactly match what is on the tape. It is permissible for the program directory to show a subset of the tape information (for example, only show the SMPMCS up to the ++JCLIN statement) if this is clearly documented.
- ☐ Rule 19.3. The cover date on the Program Directory must be updated whenever the Program Directory is refreshed for any reason and must be a higher date than the previous level of Program Directory. The cover date must be a hard-coded date (as opposed to a variable) in the Program Directory source.

Chapter 4. SYSMOD Types and Relationships

A program is made up of elements such as macros, modules, or other types of data. For SMP/E to install and service software, you must code modification control statements (MCS) for the software elements. MCS describe the elements and any relationships the software has with other software that may also be installed on the same z/OS system or subsystem. The combination of elements and MCS statements is called a system modification, or SYSMOD.

The following sections describe fundamental SMP/E concepts about SYSMODs and how they relate to product packaging.

4.1 Types of SYSMODs

Before coding any MCS for software changes, you must decide what type of SYSMOD to use. The SYSMOD type you choose depends on how the changes affect the system on which they are installed.

- A **function** introduces a new base or dependent function, or a new version or release (or both) of a function.
- A **program temporary fix (PTF)** corrects a problem that may affect all customers.
- An **APAR fix** corrects a problem that affects a specific user.
- A **user modification (USERMOD)** makes a change to an IBM product or to a user-written product.

Function SYSMODs for base functions define the environment for other SYSMODs that may be installed. All other SYSMODs are applicable to the base or dependent function that they support. For example, a PTF may fix an element that was introduced by a function SYSMOD for a particular base function--the PTF SYSMOD is therefore applicable to that function SYSMOD.

These types of SYSMODs are used to package the various software offerings already described.

- Base and dependent functions are packaged as function SYSMODs.

  Service for these functions is packaged as APAR fixes and PTFs. User-written changes for these functions are usually packaged as USERMODs.

The following sections describe the various types of SYSMODs.

Note: Refer to Chapter 10, “Naming Conventions” on page 115 for information about the required naming conventions for SYSMODs.

4.1.1 Functions

Software products can be differentiated by the type of SYSMOD, or by their relationship to other functions. The two relationships are *base* and *dependent* functions. Each of these types of functions are packaged as *function SYSMODs*.

© Copyright IBM Corp. 1986, 2003

4.1.1.1 Base Functions

A *base function* is a collection of elements (such as source, macros, modules, and CLISTs) that provides a general user function and is packaged independently from other functions.

A base function is packaged as a function SYSMOD on a RELFILE tape, identified by an *FMID*. The FMID is described under 10.2, “SYSMOD IDs” on page 115. For more information about functions, see 4.1.1, “Functions” on page 13. For more information on the implications of National Language Support on base functions, see Chapter 11, “Packaging for National Language Support (NLS)” on page 121.

Function SYSMODs for base functions are applicable to any z/OS environment, although they may have interface requirements that require the presence of other base functions.

4.1.1.2 Dependent Functions

A *dependent function* is a collection of elements (such as source, macros, modules, and CLISTs) that provides an enhancement to a base function. It may provide optional, additional function for a *base function*--this is called an "additive dependent function". A dependent function may also provide language support for a base function or for an additive dependent function--this is called a "language-support dependent function". A dependent function is identified with one, and only one, base function. On the other hand, there may be several dependent functions identified with the same base function.

A dependent function that provides language support may be for only one language. While one language may span multiple FMIDs, one FMID may not contain multiple languages. For more information, see Chapter 11, “Packaging for National Language Support (NLS)” on page 121.

A dependent function is packaged as a function SYSMOD on a RELFILE tape, identified by an FMID. FMIDs are described in 10.2, “SYSMOD IDs” on page 115. For more information about functions, see 4.1.1, “Functions” on page 13.

Function SYSMODs for dependent functions are only applicable to the parent base function. Each dependent function specifies an FMID operand on the ++VER MCS to indicate the base function to which it is applicable. (The FMID is described in 10.2, “SYSMOD IDs” on page 115.) The hierarchy defined by this relationship determines the order in which the function SYSMODs must be installed: the base function specified on the FMID operand must be installed before or concurrently with the dependent function that contains that FMID operand. For more information, see 4.2.2, “Hierarchy of SYSMOD Types” on page 18.

4.1.1.3 Choosing between Base and Dependent Functions

Depending on the needs of your product and your customers, you may need to package your product as a combination of base and dependent functions. In addition, you may need to consider how to define these functions as features for your product, as well as the charges (and terms and conditions) that will apply to your product.

You must provide Software Distribution with the appropriate building blocks (master tapes) for your product so that Distribution can ship the desired package to the customer.

Remember these points when making your decision:

Table 3. Comparison of Base and Dependent Functions

Base Function	Dependent Function
Is installable without its dependent functions. May require another base function.	Must be installed with its parent base function.
Can have no, one, or more than one dependent function	Must be associated with only one base function.
Can be explicitly deleted by another base function.	Can be explicitly deleted by another dependent function. Can be explicitly deleted by a base function.

4.1.1.4 General Packaging Rules for Functions

Packaging Rules (Functions)

- ☐ Rule 20. An z/OS installable tape cannot contain any files that will not be installed on the z/OS system. If the product can be installed on multiple operating systems, a separate tape is needed for the z/OS installable files.
- ☐ Rule 21. All elements included in an z/OS installable product must be SMP/E-installable.

Packaging Recommendations

Common elements should be packaged in a common SYSMOD.

End of Packaging Recommendations

Packaging common elements in a single SYSMOD makes it easier to service elements. When the common elements are packaged in the base function instead of in each language-support dependent function, you reduce the number of copies of that element that have to be updated when the element is serviced.

4.1.2 PTFs

A PTF SYSMOD provides *preventive service*, *corrective service*, or enhancements to a function.

Preventive service is service for a problem that a customer may not have yet encountered. By applying these PTFs, a customer may prevent problems from occurring on the system.

Corrective service is service explicitly requested by a customer to fix a problem that has occurred on the system. Corrective service is provided by a PTF distributed in response to a request for corrective service, or by a PTF. For a severe problem, an APAR fix may be provided as expedited corrective service.

A given PTF may be applicable to one or more releases of a function. Likewise, there may be more than one PTF for a given release of a function. Each PTF fixes one or more problems associated with the function.

Each PTF has a unique, 7-character name called a SYSMOD ID. PTFs are produced by Service Teams, but under certain circumstances they may also be produced by other **development** programmers.

Following are some characteristics of PTFs:

- Announcement - Service PTFs do not follow the phase cycle and are never announced.
- Licensing - PTFs are not individually licensed. If a PTF is applicable to a licensed function, it is covered by the license agreement for that function.
- Ordering - PTFs have no special order numbers. They are requested by their SYSMOD ID.
- Documentation - PTFs have no direct documentation other than a "cover letter" containing interim documentation that is part of the PTF. This cover letter describes changes or additions to a product introduced by the PTF. If a problem fixed by a PTF requires documentation corrections, the formal publications should be updated as soon as practical.

4.1.3 APAR Fixes

An APAR fix provides corrective service for a function. Corrective service is service explicitly requested by a customer to fix a problem that has occurred on the system.

An APAR fix is applicable to one, and only one, release of a function. Likewise, there may be more than one APAR fix for a given release of a function.

Each APAR fix has a unique, 7-character name. APAR fixes are produced by Service Teams, but under certain circumstances they may also be produced by other **development** programmers.

Following are some characteristics of APAR fixes:

- Announcement - APAR fixes are not announced.
- Licensing - APAR fixes are not individually licensed. If an APAR fix is applicable to a licensed function, it is covered by the license agreement for that function.
- Ordering - APAR fixes are neither ordered nor formally distributed.
- Documentation - APAR fixes have no associated documentation other than comments that may be included on RETAIN or with an APAR fix package.

Note: APAR fixes are included in a subsequent PTF.

4.1.4 USERMODs

A USERMOD is usually code written by a customer, either to change an function or to add a new function to the system. USERMODs are always applicable to a function, and require that function SYSMOD to be installed. They may also have dependencies on a PTF, APAR fix, or another USERMOD, and require other SYSMOD to be installed.

You may want to provide a sample USERMOD with your product to let customers tailor your product to their needs. For example, you may want to include a USERMOD to help your customers change or add such things as:

- A procedure in PROCLIB
- A parameter or table in PARMLIB
- A sample job in SAMPLIB
- A user exit routine

By making your product tailorable through a USERMOD, you and the user benefit from SMP/E, which does the following:

- Keeps a record of the changes
- Reports any intersections with other SYSMODs
- Makes sure the changes are not regressed
- Makes sure the changes are installed properly in the correct libraries
- Allows the users to remove the changes, if necessary

Use the ++SAMP MCS to package the USERMOD as an element for the associated function. Define the element as being installed in an appropriate data set for sample code. Use the same 7-character name for both the element in which the USERMOD is packaged and the USERMOD SYSMOD ID. (See 10.3, “Element, Alias, and Load Module Names” on page 115 for more information about element naming conventions.)

The USERMOD can be installed by the customer as is, or it can be changed before it is installed.

For examples of packaging USERMODs, see the *SMP/E User's Guide*.

Following are some characteristics of USERMODs:

- Announcement - USERMODs are not announced.
- Licensing - USERMODs are not individually licensed. If a USERMOD is applicable to a licensed function, it is covered by the license agreement for that function.
- Ordering - USERMODs are not separately orderable. They are packaged as elements of the associated product.
- Documentation - USERMODs are documented in the publications for the associated product.
- Service - USERMODs are serviced as elements of the associated product.

4.2 Defining SYSMOD Relationships

Understanding the relationships between SYSMODs is one of the most important aspects of planning how to package SYSMODs. A SYSMOD can only be installed properly and run correctly if its requirements regarding other SYSMODs on the system are met. For example, one SYSMOD may require the presence of another--a dependent function requires a particular base function. This section describes the types of SYSMOD relationships you may have to define in the course of developing and servicing a product. Specifically, it discusses the following:

- Conditional and unconditional SYSMOD relationships

- The hierarchy of SYSMOD types
- An overview of specific types of SYSMOD relationships
- Coexisting SYSMODs

4.2.1 Conditional and Unconditional Relationships

All SYSMOD relationships are either conditional or unconditional. Table 4 contrasts these two types of relationships.

Table 4. Comparison of Conditional and Unconditional SYSMOD Relationships

Conditional Relationships	Unconditional Relationships
Specified on ++IF statements	Specified on ++VER statements
Enforced if the specified function SYSMOD is present	Always enforced

4.2.2 Hierarchy of SYSMOD Types

When defining SYSMOD relationships, take into account the hierarchy of SYSMOD types. SMP/E uses this hierarchy to determine which version of an element to install, if the element is contained in several SYSMODs. Figure 2 on page 19 shows this hierarchy, from the lowest functional level to the highest.

All of the SYSMODs in the hierarchy are part of the same product version. The product version includes all SYSMODs that have the same product version in the FMID specified on their ++FUNCTION or ++VER statements. (This convention is described under 10.2, "SYSMOD IDs" on page 115.) For example, SYSMODs with these statements would be in the same product version because they all have the same product version code (MX1):

```
++FUNCTION(HMX1200).
++FUNCTION(JMX1210).
++VER(Z038) FMID(HMX1200).
++PTF(UZ10000).
++VER(Z038) FMID(HMX1200).

++FUNCTION(HMX1300).
```

For SMP/E to process SYSMODs in the correct order, you must define that order to SMP/E. As Figure 2 on page 19 shows, if a dependent function has an element in common with its base function, the element in the dependent function is used instead of the one in the base--it is functionally higher.

You define SYSMOD hierarchy with operands on the ++VER statement.

- Base functions are the lowest level in the hierarchy. Therefore, they do not specify an FMID on the ++VER statement.
- Dependent functions specify the FMID of a base function on the ++VER statement. This base function must not be for a different product.
- PTFs and APAR fixes specify the FMID of a base or dependent function on the ++VER statement.

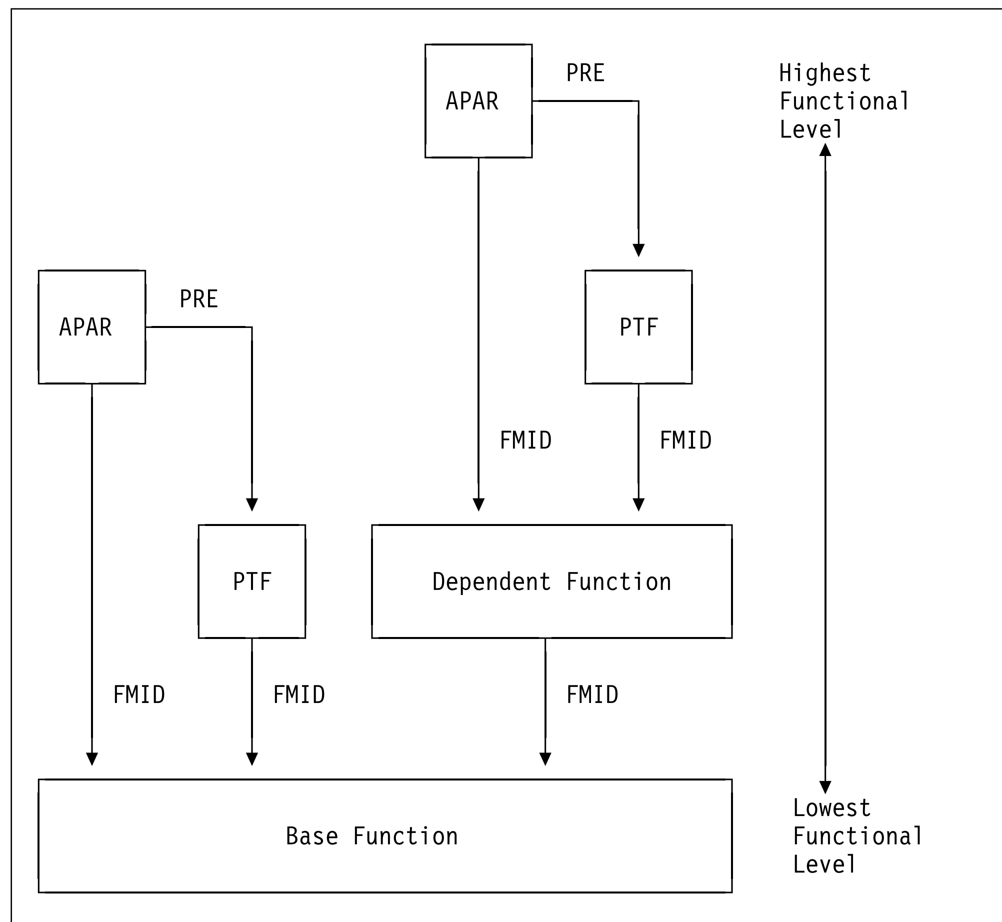


Figure 2. Hierarchy of SYSMOD Types

4.2.3 Specific SYSMOD Relationships

These are the types of relationships that may exist between SYSMODs:

- Prerequisite SYSMODs
- Corequisite SYSMODs
- Negative prerequisite SYSMODs
- Deleting and superseding SYSMODs

4.2.3.1 Prerequisite SYSMODs

Prerequisite SYSMODs have a relationship where one SYSMOD requires another.

If SYSMOD(2) needs SYSMOD(1) for proper operation, but SYSMOD(1) does not need SYSMOD(2), SYSMOD(1) is a prerequisite for SYSMOD(2). These are some cases when you would define a SYSMOD as a prerequisite:

- Defining the base function for a dependent function
- Defining the order of dependent functions
- Defining one product that is needed for another product
- Defining service for one product that is needed for another product

See 7.2.6, “Specifying Prerequisite Relationships (PRE)” on page 58 for information about how to specify this relationship, and 13.2.3, “PTF Service That Depends on Previous Service” on page 137 and 13.4.6, “Establishing the Order of Additional Dependent Functions” on page 151 for more examples.

4.2.3.2 Corequisite SYSMODs

Corequisite SYSMODs have a relationship where the two SYSMODs require each other.

If SYSMOD(2) and SYSMOD(1) need each other for proper operation, they are corequisites of each other. These are some cases when you would define SYSMODs as corequisites:

- Defining two products that need each other
- Defining two dependent functions for different products that need each other
- Defining related service for a dependent function and its parent base function

Packaging Recommendations

When you are building SYSMODs that may be installed as a group, such as prerequisite or corequisite SYSMODs, do not construct the SYSMODs in such a way that their proper installation depends on the internal processing order within SMP/E. From time to time, the processing order may be changed and SYSMODs that depend on that order may not be installed correctly. Follow the packaging rules in this book to define how the SYSMODs should be installed.

End of Packaging Recommendations

4.2.3.3 Negative Prerequisite SYSMODs

Negative prerequisite SYSMODs have a relationship where one SYSMOD requires the absence of another.

If SYSMOD(2) can be installed only if SYSMOD(1) is not also on the system, SYSMOD(1) and SYSMOD(2) are negative prerequisites. For example, you might define dependent functions as negative prerequisites if they are mutually exclusive because they tailor a product to two different environments.

Note: All negative prerequisites are unconditional and may be specified only in function SYSMODs.

See 7.2.7, “Superseding SYSMODS (SUP)” on page 59 for more information about deleting and superseding SYSMODs.

4.2.3.4 Deleting and Superseding SYSMODs

Deleting and superseding SYSMODS have a relationship where one SYSMOD replaces another.

- If other functions have a relationship with the function to be replaced, you should evaluate those relationships.
- If SYSMOD(2) takes the place of SYSMOD(1), it can delete SYSMOD(1), supersede SYSMOD(1), or both delete and supersede SYSMOD(1). For example, if Function A2 is a later release of Function A1, it can delete Function A1, supersede it, or both. The differences between deleting a SYSMOD, superseding a SYSMOD, or doing both are shown in Table 5 on page 21.

Packaging Recommendations

- A new release of a function should both delete and supersede the previous release if all of the following are true:
  - The new release contains at least all the function that was in the previous release.
  - If other products specified the deleted function as a requisite, all the internal and external interfaces used by those other products are unchanged in the new release.
  - Other products that specified the previous release as a requisite can run with the new release.
- Evaluate a replacement function using Table 5 as a guide. If the replacement function matches that description, then the preferred and recommended way to replace the previous function is to both delete and supersede it.

End of Packaging Recommendations

By both deleting and superseding the previous function, you gain the combined benefits of the DEL and SUP operands of the ++VER MCS.

- The DELETE operand is required in order to ensure that previous releases of the product are removed. This prevents accidental mixture of old and new elements in the same library.
- The SUP operand specifies that the new function completely and compatibly replaces all functions of the old function.
- The DELETE/SUP combination allows you to replace a function without disturbing any other SYSMODs that depend on that function. By specifying SUP, you are saying that the new function meets all dependencies identified by these dependent functions. You must ensure that this is the case before using the DELETE/SUP combination.

Table 5. Comparison of Deleting, Superseding, and Both Deleting and Superseding a SYSMOD

Delete	**Supersede**	**Delete and Supersede**
The new SYSMOD specifies DELETE on its ++VER statement.	The new SYSMOD specifies SUP on its ++VER statement.	The new SYSMOD specifies DELETE and SUP on its ++VER statement.
SMPCSI entries for the deleted SYSMOD are deleted. SMP/E no longer considers the deleted SYSMOD to be installed on the system.	SMPCSI entries for the superseded SYSMOD are saved. SMP/E considers the new SYSMOD to be a substitute for the superseded SYSMOD.	SMPCSI entries for the deleted and superseded SYSMOD are deleted. SMP/E considers the new SYSMOD to be a substitute for the deleted and superseded SYSMOD.
Elements for the deleted SYSMOD are deleted from the target and distribution libraries.	Elements for the superseded SYSMOD are not deleted from the target and distribution libraries.	Elements for the deleted and superseded SYSMOD are deleted from the target and distribution libraries.
Note: The new SYSMOD may replace some elements at the same or higher functional level than the deleted, superseded, or deleted and superseded SYSMOD. The new SYSMOD may also add new elements.		

Using the previous example, if no other functions have a relationship with Function A1, Function A2 can delete Function A1. On the other hand, if some other function specified A1 as a requisite, A2 should *both* delete and supersede A1. This ensures

that the requisite relationship is satisfied by both A1 and A2; no missing requisite prevents the other function from being installed.

Note: All deleting and superseding relationships are unconditional.

For specific examples of defining these relationships, see Chapter 13, “SYSMOD Packaging Examples” on page 135.

4.2.4 Coexisting SYSMODs

If two function SYSMODs can be installed in the same zone, they are said to "coexist". Two function SYSMODs can coexist if they meet all these requirements:

- They apply to the same SREL.
- Neither SYSMOD deletes nor supersedes the other.
- Neither SYSMOD is a negative prerequisite of the other.
- If the SYSMODs are base functions, they are for different products.

Packaging Rules (Zones for Product Installation)

- ☐ Rule 23.1. No product can require the customer to install it into its own unique zones, CSIs, HFS or JAR. Every product must be installable in the same target and distribution zones as any other product in the SREL, and the same HFS or JAR as any other product on the system. This gives the customer the ability to decide which combinations of products will reside together.

  The program directory may suggest or recommend that the customer use new zones, CSIs, or HFSs or JARs, but it must be clearly documented that this is optional, and installation into existing zones, CSIs, and HFSs or JARs must also be documented.
- ☐ Rule 23.2. Functions within a feature code may not specify each other on the NPRE or DELETE operand, or have any other incompatibilities that would prevent them from all being installed in the same target and distribution zones.
- ☐ Rule 23.3. If two FMIDs cannot be installed in the same zone, the SMPMCS of the FMID with the later GA date must identify the incompatibility with either an NPRE or a DELETE. If the FMIDs GA simultaneously, each FMID's SMPMCS must identify the incompatibility with either an NPRE or a DELETE for the other.

Packaging Recommendations

Make all FMIDs of a product compatible, so that the most possible function can be ordered with the fewest possible feature codes.

If a feature has multiple FMIDs, the SMP/E sample jobs provided should install them all together; products should not require FMIDs within a feature code to be installed in separate jobs.

Do not require any function or service to be ACCEPTED before another function can be APPLYed.

End of Packaging Recommendations

Although you cannot control the specific zone where a SYSMOD is installed, you can help users install the SYSMOD in the correct zone by packaging the SYSMOD correctly and by providing any additional information in the installation material. To provide this information, you must understand the rules for coexistence.

There are two ways for SYSMODs to coexist:

- Unconditionally: SYSMOD(A) is required by SYSMOD(B), and must be installed in a zone that contains SYSMOD(B). SYSMOD(A) "unconditionally coexists" with SYSMOD(B).
- Conditionally: SYSMOD(A) is not required by SYSMOD(B), and need not be installed in a zone that contains SYSMOD(B). SYSMOD(A) "conditionally coexists"with SYSMOD(B).

4.2.4.1 SYSMODs that Unconditionally Coexist

A requisite must be installed before (or concurrently with), and in the same zone as, the SYSMOD that specifies the requisite. The specifying SYSMOD cannot be installed without its requisite. Therefore, the requisite "unconditionally coexists" with the SYSMOD that specifies the requisite.

Reminder

“Unconditionally coexists with” means “must be installed before (or concurrently with) and in the same zone as.”

These are some examples of types of SYSMODs that "unconditionally coexist" with a dependent function:

- Its parent base function
- Any prerequisite dependent functions
- Any corequisite dependent functions.

4.2.4.2 SYSMODs that Conditionally Coexist

A SYSMOD that specifies a requisite is generally not required to be installed concurrently with, and in the same zone as, the SYSMOD it specifies as a requisite. Most requisites are one-way: SYSMOD(B) requires SYSMOD(A), but SYSMOD(A) does not require SYSMOD(B). The requisite (A) can be installed without the SYSMOD that needs it (B).

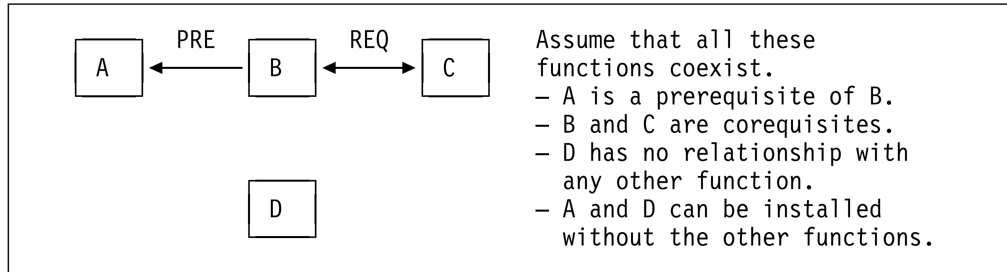
Likewise, any function SYSMODs that can coexist but do not require each other (or that do not even define any relationship to each other) are said to "conditionally coexist". They can be installed either with or without each other.

Reminder

“Conditionally coexists with” means “can be installed with but is not needed for.”

4.2.4.3 Example: Conditional and Unconditional Coexistence

Look at the following example:



Based on these relationships:

- A unconditionally coexists with B and C.
- B and C unconditionally coexist with each other.
- B and C conditionally coexist with A.
- A, B, and C conditionally coexist with D.
- D conditionally coexists with A, B, and C.

Chapter 5. Fundamental Packaging Considerations

This chapter presents the following basic considerations for product packaging:

- Installation methods for function SYSMODs
- Evaluating SYSMOD relationships
- Adding FMIDs
- Requirements for new releases
- Record length, record format, and block size requirements
- Object-code-only (OCO) requirements
- Copyright requirements
- Rework dates
- Shared libraries
- Source code requirements
- Avoiding UCLIN

Note: For any product that consists of SMP/E-installable and non-SMP/E-installable elements, that portion of the product that is SMP/E-installable must conform as much as possible to the rules and to the level of SMP/E that is available at the time of the product's GA.

5.1 Installation Methods

A product is completely SMP/E-installable when it can be installed with both of these methods:

- RECEIVE-APPLY-ACCEPT
- RECEIVE-ACCEPT BYPASS(APPLYCHECK)-GENERATE.

The installation method using GENERATE reveals problems that may occur only when the entire system is generated together; these problems may be masked in the RECEIVE-APPLY-ACCEPT scenario. Also, GENERATE is more strict about product dependencies. For example, GENERATE does not allow DD statements to override DDDEFs; as a result, duplicate library names may be exposed when a system is generated and might not be visible when an individual product is installed.

Figure 3 on page 26 provides an overview of these methods. You should plan to test both of these installation paths to ensure that the results are identical. For details on all the steps in each method, see the *SMP/E User's Guide*.

Packaging Rules (Installation)

- ☐ Rule 24. All licensed and unlicensed z/OS installable products must comply with the packaging rules for installation.
- ☐ Rule 25. If a product is not completely SMP/E-installable because of a known restriction or approved deviation, all those elements not affected by the restriction or deviation must comply with the packaging rules and the level of SMP/E that is available at the time of the product's GA.
- ☐ Rule 26. All files on a z/OS installable product tape must be SMP/E-installable.

© Copyright IBM Corp. 1986, 2003

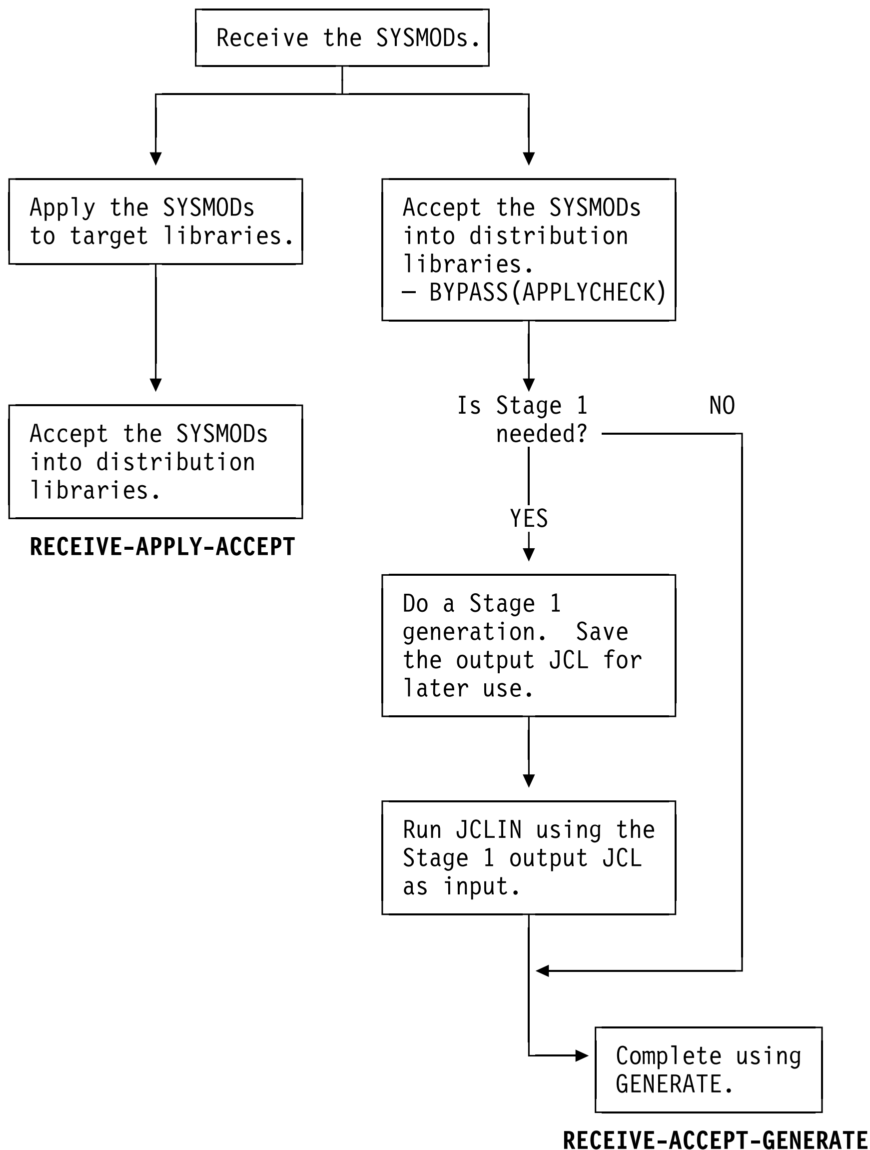


Figure 3. Overview of Methods for Installing Functions

Packaging Rules (Installation)

- ☐ Rule 26.1. A PTF must not increase its product's driving system requirements beyond what is documented in the Program Directory. It is not acceptable to require a customer to install a new Version, Release, or Mod Level of a product in order to install service.
- ☐ Rule 27. All products must be packaged so that they can be individually installed using both the RECEIVE-APPLY-ACCEPT method and the RECEIVE-ACCEPT BYPASS(APPLYCHECK)-GENERATE method.

Packaging Rules (Installation)

- ☐ Rule 27.1. The return code from ACCEPT processing for all function SYSMODs must be zero, with these exceptions:
  - Warning message GIM39701W, SYSMOD *sysmod-id* HAS NO ELEMENTS.
  - Warning message GIM50050W, concerning the DESCRIPTION operand.
  - A warning message issued only in certain environments (for example, a product tries to delete an element or load module that is not on the system).
- ☐ If a function is available in a level of SMP/E that has been available for at least a year the product should use that function instead of providing non-SMP/E jobs or tasks.
- ☐ Rule 27.3. If a function is available in a level of SMP/E that has been available for at least one year, the product must use that function instead of providing non-SMP/E jobs or tasks.

5.2 Evaluating SYSMOD Relationships

At various stages in the life of a product you must consider the SYSMOD relationships you may have to define:

- **Packaging the initial release of a function**

  The first time you package a product as a function, you must consider whether that function has any dependencies on other SYSMODs that might be installed on the same system.

  Some SYSMODs have no relationships to any other SYSMODs. For example, you may have a simple function SYSMOD that stands on its own and has no requirements for any other functions to be installed. On the other hand, you may have a function that uses code provided by another function.

  If a SYSMOD stands on its own, you do not have to define any relationships. However, if there are requirements for other SYSMODs, you must define these relationships.

  Refer to 4.2, “Defining SYSMOD Relationships” on page 17 for more information.

- **Replacing a function**

  After a function has been available for a while, you may develop enough changes to distribute a new release of that function. You must define the relationship of that new release to the previous release, as well as decide whether to carry over relationships defined in the previous release.

  See 4.2.3.4, “Deleting and Superseding SYSMODs” on page 20 for more information.

- **Enhancing a function**

  When you have changes for a function, you may want to update the function instead of replacing it.

For example, you may want to provide some optional capability for a particular environment. These enhancements could be packaged as a dependent function, which adds to the base function without replacing it. You must define the relationship between the base function and its dependent functions. If you develop several dependent functions, you must also define any relationships there may be among them.

- **Supporting language-sensitive elements**

  When a product provides language-sensitive elements (such as messages and dialog elements), those language-sensitive elements must be packaged in a separate dependent function for each language, including U.S. English (even if U.S. English is the only language supported). The remaining elements remain in the base function. You must define the relationship between the base function and its dependent functions for language-sensitive elements.

  Refer to Chapter 11, “Packaging for National Language Support (NLS)” on page 121 for more information.

- **Servicing a function**

  At each of the stages in developing a function you may need to provide service to fix problems with the function. You must define the relationship between the service and the function. As you provide more service, you may also have to define relationships among service fixes.

  Refer to 4.1.2, “PTFs” on page 15 and 4.1.3, “APAR Fixes” on page 16.

5.3 Adding FMIDs

New FMIDs are required for functions in new versions, releases, or modification levels. You need to be aware of the requirements for adding FMIDs.

Packaging Rules (++FUNCTION SYSMOD ID)

- □ Rule 29. If new function is introduced in any manner other than in an APAR or PTF, a new FMID is required for the new function. If an existing function is modified in any manner other than through the integration of an APAR or PTF, a new FMID is required for the updated function.
- □ Rule 30. When you issue a new release of a base function you must either merge and integrate any additive dependent functions for the previous release, or repackage and reissue new releases of those dependent functions. If you repackage and reissue a new release of a dependent function, you must use a new FMID.

Packaging Recommendations

- Use the smallest possible number of FMIDs. Consider combining FMIDs in cases such as:
  - The FMIDs are always shipped and installed together.
  - The base and the English NLV can be shipped as one FMID.
  - The FMIDs are always on the same development schedule.

- Avoid combining FMIDs with different driving system requirements or different export restrictions.

End of Packaging Recommendations

5.4 Requirements for New Releases

Following are the rules for new releases of base and dependent functions. For more information, see 4.1.1, “Functions” on page 13.

Packaging Rules (New Releases of Base and Dependent Functions)

- ☐ Rule 31. A new release of a base function must not supersede any PTF or APAR fix incorporated into or designed out of that release except in the following cases:
  - Following APAR cutoff, the new release has included additional fixes for APARs that were not committed to be fixed. The new release must supersede any PTFs that were committed for fixing those APARs, since they are already part of the new release. This downlevel control ensures that unnecessary PTFs will not be built, and avoids installation problems that could result if those PTFs are required by any other functions.
  - PTFs or APAR fixes applicable to the new release are available to customers, but were also incorporated into the new release before it was made available. The new release must supersede these PTFs and APARs to prevent them from being installed, since they are already part of the new release.

Packaging Rules (New Releases of Base and Dependent Functions)

- ☐ Rule 32. A new release of a **dependent** function must supersede any PTF or APAR fix incorporated into or designed out of that release in the following cases:
  - The PTF or APAR fix is applicable to the parent base function and is not a prerequisite (++VER PRE) or corequisite (++VER REQ) of the new release.
  - The PTF or APAR fix is applicable to a different, coexisting dependent function that is applicable to the same parent base function, and that PTF or APAR fix is not a prerequisite (++VER PRE) or corequisite (++VER REQ) of the new release.

    Note: A coexisting dependent function is one that is not deleted (or deleted and superseded) by the new release of the dependent function.
  - Following APAR cutoff, the new release has included additional fixes for APARs that were not committed to be fixed. The new release must supersede any PTFs that the Service Team had committed for fixing those APARs, since they are already part of the new release. This downlevel control ensures that unnecessary PTFs will not be built, and avoids installation problems that could result if those PTFs are required by any other functions.
  - PTFs or APAR fixes applicable to the new release are available to customers, but were also incorporated into the new release before it was made available. The new release must supersede these PTFs and APARs to prevent them from being installed, since they are already part of the new release.

    Note: If this new release of the dependent function names as a prerequisite the last PTF in a supersede chain (using ++VER PRE), the new release does not have to supersede any of the other PTFs in that chain.

5.4.1 Consolidating Functions and Service for Elements

Every so often, it may be necessary to consolidate function and service for the elements of a particular product release. There are two ways to merge and integrate the elements:

- Create a service update for the existing release of the product.
- Create one or more new functions that explicitly delete their predecessors. This type of consolidation is described below.

Consider consolidating element function and service into a new version or release in these cases:

- The number of requisites being managed by SMP/E and the customer for the existing elements' function and service is very large, and the SYSMODs have complex relationships.

- Numerous and complex requisites degrade performance when the function is being installed.
- The number of element versions being maintained by **service** for the existing functions and service is very large.
- The code that must be recompiled by **development** for the consolidation is a significant portion of the product's elements. Code may have to be recompiled because of required element changes, such as changed FMID flagging, changed licensing and copyright indicators, changes in distribution classification, and so on.

5.5 Record Length, Record Format, and Block Size Requirements

You should take these requirements into consideration as you develop your product; this will minimize problems when you build your RELFILE tape. Table 6 on page 33 summarizes the requirements for packaging elements in RELFILEs.

Packaging Rules (Macros, Modules, Source)

- ☐ Rule 33. Unless otherwise stated in the product's announcement materials, every product must be installable on all supported DASD. For example, the block size of a dataset may not exceed the maximum allowable block size on any supported DASD type.
- ☐ Rule 33.1. Jobs allocating target or distribution libraries must specify BLKSIZE=32760 for all RECFM=U datasets, and BLKSIZE=0 (utilizing system-determined blocksizes) for all non-RECFM-U datasets, with the following exceptions:
  1. SYS1.UADS
  2. Font libraries
- ☐ Rule 34. Macros, modules, and source elements must be members of a partitioned data set (DSORG=PO).
- ☐ Rule 34.1. Distribution libraries must be partitioned; only target libraries may be sequential. Use of sequential distribution libraries would tend to increase the total number of datasets required on the system.
- ☐ Rule 35. The record format (RECFM) for load modules must be U.

  This rule replaces rule 41. For more information, see Table 6 on page 33.
- ☐ Rule 35.1. A product must not change any of the following attributes of an existing dataset:
  - RECFM
  - PDS vs. PDS/E vs. Sequential
  - LRECL
  - PATH attributes

  If such a change is required, a new dataset must be created, which must have a new DDDEF entry and a new DDNAME and dataset name (which must adhere to the dataset naming rules).

Packaging Restriction

Restriction 8. SMP/E requires the record format (RECFM) for macros and source to be FB, and the record length (LRECL) to be 80. For more information, see Table 6 on page 33.

End of Packaging Restriction

Packaging Recommendations

- A dependent function should contain only those elements needed to provide the additive function, or that are needed to provide the additional language support. This reduces the number of element versions and makes servicing the elements easier.
- Sample job streams and other special data that might be helpful to the customer can be stored as a member of a partitioned data set that is unloaded to a relative file on the RELFILE tape. Examples include:
  - A procedure to allocate and catalog libraries
  - Installation verification procedures (IVPs)

  This data should be packaged as sample code using the ++SAMP MCS, and it should be defined to be installed in an appropriate data set for sample code.

  When SMP/E installs the SYSMOD, it copies this member into the libraries specified by the SYSLIB and DISTLIB operands on the element MCS. The sample job stream or other data can then be retrieved from the appropriate library for further processing.

End of Packaging Recommendations

Packaging Rules (Data Elements and HFS and JAR Elements)

- ☐ Rule 37. Data elements, HFS and JAR elements, and ++PROGRAM elements must be packaged as members of a partitioned data set (PDS) or partitioned data set extended (PDSE) (DSORG=PO).
- ☐ Rule 38. The record format (RECFM) of data elements, HFS, and JAR elements must be F, FA, FM, FB, FBA, FBM, V, VA, VM, VB, VBA, or VBM. The record format (RECFM) of ++PROGRAM elements must be U.

  Notes:

  1. Elements with fixed-length records are not restricted to a logical record length (LRECL) of 80.
  2. A VSAM data set may be a data element if it is in AMS REPRO format. However, after the data is installed by SMP/E, the customer will also have to run an AMS REPRO job to create the original form of the VSAM data. (SMP/E does not support native VSAM data sets as elements.)
- ☐ Rule 39. Elements with variable-length records may not contain spanned records.

Packaging Rules (Data Elements and HFS and JAR Elements)

- ☐ Rule 39.1. CLISTs and EXECs must not have sequence numbers.

Packaging Recommendations

- If a function SYSMOD uses unique target or distribution libraries, you may want to include a procedure to allocate and catalog the libraries. This procedure should be in an appropriate data set for sample code, must be a member in one of the relative files, and must be defined by the appropriate element MCS, as described above.
- A product may have an Installation Verification Procedure (IVP) that may be used by customers to verify that the product has been installed. If an IVP is included in the product package, it should be in an appropriate data set for sample code, must be a member in one of the relative files, and must be defined by the appropriate element MCS.

End of Packaging Recommendations

Table 6 (Page 1 of 2). Summary of Requirements for Packaging Elements in RELFILEs

Element Type	MCS	RECFM	LRECL	Recommended BLKSIZE
Macro	++MAC	FB	80	8800 (See note 1.)
Module (Each object module must be link-edited into a single-module load module.)	++MOD	U	No specific required value	6144 (See note 1.)
Source	++SRC	FB	80	8800 (See note 1.)
Data element (See note 2.)	++*element*	No specific required value	No specific required value	For FB80, use BLKSIZE 8800. For other formats, BLKSIZE depends on DASD. (See note 1.)

Table 6 (Page 2 of 2). Summary of Requirements for Packaging Elements in RELFILEs

Element Type	MCS	RECFM	LRECL	Recommended BLKSIZE
HFS or JAR element (See note 2.)	++HFS or ++JAR	No specific required value	No specific required value	For FB80, use BLKSIZE 8800. For other formats, BLKSIZE depends on DASD. (See note 1.)

Notes:

1. Use the most efficient block size for the DASD you support. The block size must not exceed that of the smallest DASD supported by your product, as indicated in the product's documentation. If the smallest DASD supported is the 3350, the block size must not exceed 19069.
2. Data elements, HFS, and JAR elements (unlike macros, modules, and source) have no required record format or logical record length.

5.6 Specifying Copyright Information

Copyright information is legal requirement for both the program directory and the product tape. It is not required that the copyright date in the program directory match the date on the ++FUNCTION statement. The date on the program directory reflects the date of publication, and the date in the MCS represents the date of the product code.

5.6.1 Program Directory (Installation Manual)

The program directory has a copyright date on the cover page.

5.6.2 Product Tape

A copyright comment is required for all licensed programs. This is done with the ++FUNCTION statement. See 7.1.4, “Specifying Copyright Information” on page 53 for copyright rules and an example.

5.7 Specifying a Rework Date

The rework date is used by SMP/E to ensure that down-level product code is not installed on a system.

For more information, see 3.2.1, “Contents of the Program Directory” on page 11 and 7.1.2, “Identifying the REWORK Date (REWORK)” on page 50.

5.8 Shared Libraries

This section discusses shared libraries. For information about library names, refer to 10.4, "Library Names" on page 117.

Packaging Rules (Shared Libraries)

- ☐ Rule 42. A library cannot contain two or more elements with the same name or alias name, even if they are different types. Therefore, if your product is to be installed in libraries shared with another product, you must ensure that none of your product's elements have the same name or alias name as those for elements of the other product that are installed in the same library.
- ☐ Rule 43. If products share a library, each product must use the same data set attributes for that library. This means that if a product adds elements to an existing product-specific library, the new product must specify the same DCB attributes as the existing library.
- ☐ Rule 43.1. If Product A adds, deletes, or changes members of a library allocated by Product B, then Product A must identify Product B as "required for install" in its Program Directory.

If different products contain like-named elements (or aliases), data can be overlaid; this can produce unpredictable results.

If products share a library but specify different data set attributes, installation errors can occur.

If products share a library, the products must make sure that there will be sufficient space left in the library after installation. The library must be able to fit on all DASD types supported by all the products. Also, there must be sufficient space remaining so that the products can be serviced.

Packaging Recommendations

To avoid problems with like-named elements or aliases, do not install your product in shared libraries.

End of Packaging Recommendations

5.9 Source Code

Source code and any associated macros may be provided for a product that is not totally object-code-only (OCO). For a new release of a base function, a product may provide all the non-OCO source code and macros associated with that base function. For a new release of a dependent function, a product may provide only the non-OCO source code and macros that are added or changed by that dependent function.

The non-OCO source code and macros may be provided in one of the following ways:

- Basic materials source code - Source code and any associated macros may be provided to customers on the basic materials RELFILE tape.

  The source code and macros may be packaged as ++SRC and ++MAC elements respectively on relative files of the product's basic materials RELFILE tape.

  Note: SMP/E invokes **only** the assembler to process source code. If any other kind of source code is packaged as ++SRC elements, the corresponding ++MOD must be present to prevent SMP/E from attempting to process the source through the assembler.

- Optional materials source code - Source code and any associated macros may be provided to customers on an optional materials source code tape. The source and macros are not packaged with SMP/E MCS, are not serviceable, and are not supported by Custom-Built offerings.

  By convention, any macros should precede the source code on the tape. The source code and macros must be in IEBUPDTE or IEBCOPY format on the tape.

Also refer to 5.6, "Specifying Copyright Information" on page 34 and 7.1.4, "Specifying Copyright Information" on page 53 for additional information.

5.10 Avoiding UCLIN

UCLIN can cause many complications and must be avoided. Some potential problems resulting from UCLIN are:

- Increased chances for introducing errors
- Difficulty in debugging errors
- Performance impact for customized offerings

Packaging Recommendations

Do not use UCLIN. Use MCS instead.

Note: UCLIN is acceptable, and recommended, to create or modify DDDEF entries.

End of Packaging Recommendations

MCS can:

- Add modules to existing load modules
- Change ownership of an element
- Move macros, modules, source, and load modules
- Rename load modules
- Delete load modules
- Delete elements

Table 10 on page 71 describes some things you can do through MCS to avoid using UCLIN.

Refer to Chapter 8, "Using MCS to Manipulate Elements and Load Modules" on page 71 for more information.

Chapter 6. Elements and Load Modules

The term "element" is used as a collective name for such things as:

- Source
- Macros
- Modules
- CLISTs
- Panels
- Procedures
- Sample programs that make up a product

Each element is distributed under a unique name (starting with the 3-character prefix for the product and referred to as the *element name*) and performs some particular function for the product that owns the element.

The element statements describe elements contained in a function. All elements of a product on the product tape must be described in its MCS. SMP/E provides a variety of MCS to accommodate a broad spectrum of element types, including language-sensitive versions of many element types.

- ++MAC describes a new or replacement macro.
- ++MOD describes a new or replacement module (a single-CSECT load module).
- ++SRC describes a new or replacement source module.
- Data element MCS describe new or replacement elements that are not macros, modules, or source. See 6.4, “Data Element Types” on page 39 for a list of the element data types.
- ++HFS and ++JAR describes a new or replacement element that is installed in a hierarchical file system (HFS) or Java Archive (JAR) file.

This chapter describes various considerations for packaging the elements that make up a product. These topics are discussed:

- Element ownership
- Element aliases
- Data element types
- Load modules
- Generation macros
- Packaging sample JCL and data
- Language sensitive elements

6.1 General Packaging Rules, Restrictions, and Recommendations for Elements

This section describes general rules for packaging elements.

© Copyright IBM Corp. 1986, 2003

Packaging Rules (Elements)

- ☐ Rule 47. Any source code or macros delivered to the customer must not contain any security classification, including:
  - *OCO*
  - *CONFIDENTIAL*
  - Object Code Only

  where * equals a blank or special nonalphanumeric character.
- ☐ Rule 48. If you plan to update any ++SRC or ++MAC elements by **service**, they must have sequence numbers. (Because data elements cannot be updated, sequence numbers are not required.)
- ☐ Rule 48.1. The FROMDS operand is not permitted on the ++PROGRAM statement.
- ☐ Rule 49.1. An element can be owned by only one function. Ownership is defined by the FMID and VERSION operands on the ++VER and element statements.

Packaging Recommendations

- Single-CSECT modules are recommended where possible. This makes it easier for the module to be serviced. A single CSECT can be distributed rather than shipping the entire module. SMP/E can perform a CSECT replacement.
- If the ultimate destination of an element is PARMLIB or PROCLIB, and it is copied there without modification, the product should install it directly into PARMLIB or PROCLIB. It is the product developer's responsibility to ensure that the element name is unique within the library.

End of Packaging Recommendations

For more details, see Chapter 8, “Using MCS to Manipulate Elements and Load Modules” on page 71.

6.2 Element Ownership

An element must be exclusively owned by one product. For guidelines on moving an element from one product to another, see 13.8, “Changing the Contents of Products” on page 160.

You may have an element that is owned by one particular product and is being shared between your product, the owning product, and other products that have already been designed and delivered. In that case, for any new release of the owning product, your product (and each other sharing product) may need to provide either a PTF or a new release to ensure that all these products can still be installed.

6.3 Using Aliases for Elements

An *alias* is an alternative name assigned to an element or load module. It is important to maintain the uniqueness of these names to:

- Ensure that pieces are not unintentionally overlayed
- Make each entity identifiable to its owning product
- Allow each piece of a product to be serviced

For z/OS products, an element is defined in the SMPMCS file using the ++*element(ccccxxxx)* statement, where *ccccxxxx* is the name assigned to that particular element. Each piece that is shipped on a product tape must be defined in the SMPMCS file as either an element or an alias of an element. If an alias name is assigned to an element, the RELFILE tape must contain both the element and the alias in a RELFILE.

Refer to 10.3.3, "Alias Names" on page 117 for information about alias names.

6.4 Data Element Types

Table 7 lists the MCS that can be used to define data elements. It may not reflect the most currently supported values. For the latest information, see the *SMP/E Reference* manual.

Table 7 (Page 1 of 2). MCS for Data Elements. If an element is provided in only one language, the x's can be left off the MCS. If an element is provided in more than one language, replace the x's with the appropriate value from Table 13 on page 123.

MCS	Description
++BOOK*xxx*	Online book member
++BSIND*xxx*	Index for an online publications library (bookshelf)
++CGM*xxx*	Graphics source for an online book
++CLIST	CLIST
++DATA	Data not covered by other types
++DATA1–++DATA5	IBM generic data types 1–5 These are for IBM use only, to define elements that are not covered by any existing data types.
++DATA6*xxx*	IBM generic data type 6 This is for IBM use only to define an element not covered by any existing data types.
++EXEC	EXEC
++FONT*xxx*	Printer Font Object Contents Architecture (FOCA) font
++GDF*xxx*	GDF graphics panel
++HELP*xxx*	Help information (for example, a member in SYS1.HELP or a dialog help panel)
++IMG*xxx*	Graphics image for an online book
++MSG*xxx*	Message member (such as for a dialog or for a message data set)
++PARM	PARMLIB member

Table 7 (Page 2 of 2). MCS for Data Elements. If an element is provided in only one language, the x's can be left off the MCS. If an element is provided in more than one language, replace the x's with the appropriate value from Table 13 on page 123.	
MCS	**Description**
++PNL*xxx*	Panel for a dialog
++PROBJ*xxx*	Printer object element
++PROC	Procedure in PROCLIB
++PRSRC*xxx*	Printer source element
++PSEG*xxx*	Graphics page segment for an online book
++PUBLB*xxx*	Online publications library (bookshelf)
++SAMP*xxx*	Sample data, program, or JCL in a data set for sample code
++SKL*xxx*	File skeleton for a dialog
++TBL*xxx*	Table for a dialog
++TEXT*xxx*	PDS member containing text plus SCRIPT tags
++USER1–++USER5	User-defined data types 1–5 These are for user-defined elements that are not covered by any existing data types.
++UTIN*xxx*	General utility input
++UTOUT*xxx*	General utility output

Some types of elements, such as panels, messages, or text, may have to be translated into several languages. In these cases, the corresponding MCSs contain *xxx* to indicate which language is supported by a given element. Refer to 11.1, "Element Types for Translated Data Elements" on page 122 for a description of national language identifiers. Figure 4 on page 41 shows an example where product XX1 (with a component code of ZZZ) must provide both English and French support for a message module, a panel, a panel message, and a sample element.

Notes:

1. The message modules can be in the same distribution library, because the element names are different.
2. For the panels, dialog messages, and samples, there is a different element *type* for each language version of an element. Therefore, the element : GIM99XMP for all the languages in which the element is supported. However, elements with the same name must be installed in different libraries. (SMP/E does not check whether different types of data elements have the same name. Likewise, SMP/E does not prevent elements with the same name from being installed in the same libraries.)

```
++FUNCTION(FXX1101).                ++FUNCTION(FXX1102).
++VER(Z038) FMID(EXX1100).          ++VER(Z038) FMID(EXX1100).
++MOD(ZZZMOD0E)...       message    ++MOD(ZZZMOD0F)...
  DISTLIB(AZZZMOD1).      modules     DISTLIB(AZZZMOD1).
++PNLENU(ZZZPNL01)...    panels     ++PNLFRA(ZZZPNL01)...
  DISTLIB(AZZZPNLE)                   DISTLIB(AZZZPNLF)
  SYSLIB(SZZZPNLE).                   SYSLIB(SZZZPNLF).
++MSGENU(ZZZMSG01)...    dialog     ++MSGFRA(ZZZMSG01)...
  DISTLIB(AZZZMSGE)       messages    DISTLIB(AZZZMSGF)
  SYSLIB(SZZZMSGE).                   SYSLIB(SZZZMSGF).
++SAMPENU(ZZZSMP01)...   samples    ++SAMPFRA(ZZZSMP01)...
  DISTLIB(AZZZSAME)                   DISTLIB(AZZZSAMF)
  SYSLIB(SZZZSAME).                   SYSLIB(SZZZSAMF).
```

Figure 4. Example of Using Data Element MCSs

6.4.1 USERx Data Types

User-defined data types are for user-defined elements that are not covered by any existing data types. Therefore, USER*x* data types are reserved for end users.

6.5 Shared Load Modules

Sometimes, products need to share load modules.

- A load module can contain multiple modules, some of which are owned by different FMIDs. This is called a *shared load module*. Examples of shared load modules are load modules that contain:
  - Compiler or high-level language (HLL) modules
  - Callable system services
  - Subsystem or product interfaces (for example, CICS, DB2, ISPF)
  - Modules from base and dependent functions or multiple dependent functions.
  - Modules from different products
- A module that can be link-edited into more than one load module or be dynamically accessed by more than one load module. This is called a *shared module*. Examples of shared modules are:
  - Compiler or high-level language (HLL) modules
  - Callable system services
  - Subsystem or product interfaces (such as CICS, DB2, and ISPF)
  - Modules being *reused* by more than one load module

Note: Refer to 10.3, "Element, Alias, and Load Module Names" on page 115 for more information about load module names and to 8.5, "Enabling Load Module Changes at the CSECT Level (++MOD CSECT)" on page 77 for information about using the ORDER statement for load modules.

Packaging Rules (Shared Load Modules)

- ☐ Rule 52. One product must not use JCLIN to redefine the content of another product's load module (even for shared load modules). For more information, see 6.1, "General Packaging Rules, Restrictions, and Recommendations for Elements" on page 37.
- ☐ Rule 54. Ensure that the product owner of a module that is shared across products does not use ++MOD DELETE and that it does not change the SYSLIB or DISTLIB of the shared module. For more information about deleting load modules, see 8.3, "Deleting Load Modules (++DELETE)" on page 75.
- ☐ Rule 55. If a module in Product A requires elements from Product B and the products are installed in different zones, the program directory for Product A must define Product B as a prerequisite.

Packaging Recommendations

- If a shared module is loadable and is used by more than one product, then products that share modules should dynamically load the shared modules during initialization and then link (or branch) to it as needed (there are performance considerations). This way, the latest level of the module is used without having to link-edit the module every time it is serviced.
- If a module is link-edited into a known existing load module and does not require link edit control statements (such as ENTRY, ALIAS, and ORDER), the ++MOD LMOD operand should be used instead of shipping JCLIN.

End of Packaging Recommendations

For more information about sharing load modules, see 9.6.4, "Cross-Product Load Modules for Products Installed in the Same Zone" on page 100 and 9.6.5, "Cross-Product Load Modules for Products Installed in Different Zones" on page 102.

6.6 Generation Macros

Macros for system, subsystem, and product generation are packaged and distributed with the products that own them. However, they may be used to install a variety of other products.

Generation macros require that steps be taken to ensure that the correct level is maintained if the *owning* macro reships the macro; this could cause down-leveling for another product that may have shipped a different level of the same macro.

If your product requires changes in a generation macro owned by another product, you must negotiate with the owner of the other product for the required changes.

The function of system generation macros can be better performed through JCLIN. For more information about special generation macros, see 9.1, "Providing JCLIN Data for Function SYSMODs" on page 81.

Packaging Rules (Generation Macros)

- ☐ Rule 56. New products must not use generation macros.

Packaging Recommendations

Existing products should not introduce new generation macros.

End of Packaging Recommendations

6.7 Sample JCL and Data

Sample JCL and data for a product may be stored as a member of a partitioned data set (PDS), then packaged in a relative file on a RELFILE tape for a function SYSMOD.

- To package sample JCL, use the ++SAMP statement.
- To package sample data, use either the ++SAMP statement or the ++DATA*n* statement.

For both sample JCL and sample data, make sure to specify an appropriate data set for sample code.

When SMP/E installs the function SYSMOD, it copies the elements into the libraries specified by the SYSLIB and DISTLIB operands on the MCS. The sample JCL or data can then be retrieved from one of these libraries for further processing.

Packaging Rules (Samples)

- ☐ Rule 58. Every product must ship a sample job to allocate any target or distribution libraries that are created by the product, and must require the installer to run it. If any of these libraries are shared libraries that may have been allocated by other products, such libraries are to be allocated in a separate job or job step, with instructions to the user explaining when the job or job step is to be run.

  Every product must ship a sample job to create DDDEF entries for new target and distribution libraries, as well as any existing libraries that may not have entries in this product's target or distribution zone. Entries for all distribution libraries must be created in the distribution zone, and entries for all target libraries must be created in the target zone. In addition, entries for all distribution libraries should be created in the target zone to support RESTORE processing.

  If you install into another product's library, you must ensure that DDDEF entries for that library exist in your zone. There are two ways to do that:

  1. Require the presence of the other product in the zone, using the REQ, PRE, or FMID operand on the ++VER statement. (Simply documenting it in the Program Directory is not enough.)

     Note: SMP/E-enforced requisites should only be used if necessary for the installation.

  2. Create the entry yourself in your DDDEF job, using the ADD DDDEF statement to prevent overlaying the existing DDDEF entry if it happens to be in your zone.

     Recommendation: The ADD DDDEF statements for these libraries should be separated from the other ADD DDDEF statements, with instructions to the installer that they should not be used if the DDDEF entries already exist.

  The program directory must identify the names of the sample jobs and in which RELFILE they reside, so that customers can download the jobs directly from the tape, if desired. The program directory will also state that the customer can perform an SMP/E RECEIVE to load the jobs into temporary libraries, copy them into private data sets, and then modify and run the jobs from these data sets.

  If Function A requires Function B with the FMID, PRE, or REQ operands, and Function A uses Function B's libraries, then Function A is not required to ship allocation or DDDEF jobs for any libraries allocated by Function B.

  Exception: The following z/OS BCP data sets can be assumed to always exist, and therefore need not be allocated in product allocation jobs:

```
ABLSCLI0    AKHELP      AOSXCF      MIGLIB
ABLSKEL0    ALINKLIB    AOS00       MODGEN
ABLSMSG0    ALPALIB     AOS06       MSGENU
ABLSPNL0    AMACLIB     AOS11       MSGJPN
ABLSTBL0    AMIGLIB     AOS12       NUCLEUS
ACBDCLST    AMODGEN     AOS32       PARMLIB
ACBDHENU    AMSGENU     APARMLIB    PROCLIB
ACBDHJPN    AMSGJPN     APROCLIB    SAMPLIB
ACBDMENU    ANUCLEUS    ASAMPLIB    SBLSCLI0
ACBDMJPN    AOSACB      ATSOMAC     SBLSKEL0
ACBDMOD1    AOSBA       CMDLIB      SBLSMSG0
ACBDMOD3    AOSBN       CSSLIB      SBLSPNL0
ACBDPENU    AOSB0       DGTPKLB     SBLSTBL0
ACBDPJPN    AOSB3       HASPSRC     SCBDCLST
ACBDTENU    AOSCD       HELP        SCBDHENU
ACBDTJPN    AOSCE       IMAGELIB    SCBDHJPN
ACMDLIB     AOSC5       JES3LIB     SCBDMENU
ACSSLIB     AOSD0       JES3MAC     SCBDMJPN
ADGTPKLB    AOSG0       KHELP       SCBDPENU
AGENLIB     AOSH1       LINKLIB     SCBDPJPN
AHELP       AOSH3       LPALIB      SCBDTENU
AJES3MAC    AOST3       MACLIB      SCBDTJPN
AJES3SRC    AOST4
```

This rule is continued on the next page.

Packaging Rules (Samples)

Rule 58 (Continued)

You should supply a DDDEF and dataset allocation job and you should comply with the following:

1. The following lines should appear in the instructions for the allocation job, where xxxxxxxx is the name(s) of the corresponding DDDEF job(s):

```
//*    If you specify a volume for any dataset in this job, you
//*    must also specify the same volume in the corresponding
//*    DDDEF entry in the DDDEF job, xxxxxxxx.
```

2. The following lines should appear in the instructions for the DDDEF job, where xxxxxxxx is the name(s) of the corresponding allocation job(s):

```
//*    If you specify a volume for any dataset in this job, you
//*    must also specify the same volume in the corresponding
//*    dataset allocation job, xxxxxxxx.
```

3. Allocation jobs must not use JCL referbacks for the DCB attributes of any dataset; the attributes must be specified explicitly for each dataset. This rule improves readability and usability of the jobs, and permits automated validation of the dataset allocations.

4. If the DDDEF job includes DDDEFs for paths in the HFS or JAR, the last step of the DDDEF job must be exactly as follows:

```
//**********************************************************
//*  Change the -PathPrefix- string to the appropriate     *
//*  high level directory name.  If you are installing in   *
//*  the path as defined, change "-PathPrefix-" to ""       *
//*  (null).  If you are upgrading releases or installing   *
//*  maintenance, change "-PathPrefix-" to "/Service" or a  *
//*  more meaningful name.  Please note that the            *
//*  replacement string is case sensitive.                  *
//*                                                         *
//*  Please verify that the changed path statements do not  *
//*  contain double slashes (such as //usr/lpp) prior to    *
//*  running this step.                                     *
//**********************************************************
//DEFPATH EXEC PGM=GIMSMP,REGION=4096K
//SMPCSI  DD   DSN=#globalcsi,
//             DISP=SHR
//SMPCNTL DD *
    SET BDY(#tzone) .         /* change -PathPrefix- */
      ZONEEDIT DDDEF.
      CHANGE PATH('/usr/lpp/@abc@'*,
                  '-PathPrefix-/usr/lpp/@abc@'*).
    ENDZONEEDIT.
/*
```

where @abc@ is enough of the pathname to make the change succeed.

- ☐ Rule 58.1. Products must not ship catalogued or instream procedures to invoke SMP/E during installation. Sample installation jobs must invoke SMP/E directly, and must require the installer to create DDDEF entries for all libraries.

- ☐ Rule 58.2. The required DDDEF and allocation jobs, as well as any other supplied jobs that invoke SMP/E, may require editing prior to submission, but must not require a specific editor or use of any specific dialog, PROC, or batch update job as the sole supported path for SMP/E installations. Every product must provide and document an "edit and submit" sequence for its install jobs.

- ☐ Rule 58.3. Jobs that invoke SMP/E or allocate target or distribution libraries must be shipped intact; jobs or CLISTs that generate these jobs must not be shipped.

Packaging Rules (Samples)

- ☐ Rule 58.4. Sample jobs must not use UNIT=VIO.

- ☐ Rule 58.5. Sample allocation jobs must specify UNIT=SYSALLDA for all target and distribution libraries.

- ☐ Rule 58.6. Sample DDDEF creation jobs must specify UNIT(SYSALLDA) for all target and distribution libraries.

- ☐ Rule 58.7. All products installing into the HFS or JAR must statically create their directories in a MKDIR exec. The Program Directory must document how to run the exec during the installation of the product, similar to the documentation on running dataset allocation jobs.

  The MKDIR exec must meet the following requirements:

  1. It accepts a parameter for the highest-level directory, rather than hard-coding it.
  2. Output is sent to the SYSOUT held queue. It contains a report of what was created, what was not created, and what directories already existed. It also includes the return code received and the date and time it was run.
  3. The directory names all appear together.
  4. It can be executed multiple times successfully, before or after APPLY processing.

     NOTE: Unlink commands must be coded so that they will preserve the integrity of any symlinks or files created by the SMP/E APPLY.

- ☐ Rule 58.8. If a product installs into the HFS or JAR, its DDDEF job must include DDDEFs for the /usr/lpp/xxxxxxxx/zzzzzzzz/ directories it installs into, where xxxxxxxx is one or more subdirectory names. The DDDEF job must create the pathname in the DDDEF, and then provide a separate step to edit the DDDEF and change the path to the user-defined prefix. This is necessary to accommodate long pathnames that are not easily edited by hand.

- ☐ Rule 58.9. Products must not create DDDEF entries for any path other than /usr/lpp/xxxxxxxx/IBM/, where xxxxxxxx is one or more subdirectory names.

- ☐ Rule 58.10. If a product provides EXECs run during installation, such as MKDIR EXECs, a batch job invoking the EXEC must be provided for the customer's use. The EXEC and the batch job must be two different members; the EXEC must not be imbedded in the job.

  This does not apply to EXECs run after installation, such as IVPs or customization.

- ☐ Rule 58.11. Products are not permitted to use SMP/E's dynamic allocation function to allocate target and distribution libraries as new data sets; the usage of DDDEFs is only acceptable after the datasets have been allocated outside of SMP/E.

- ☐ Rule 58.12. Every dataset identified in the allocate job must appear in the appropriate Storage Requirements Table in the Program Directory, and the RECFM and LRECL for that dataset in each location must match. Every HFS or JAR path appearing in a DDDEF job must appear in the HFS and JAR Paths Table in the Program Directory.

-

- ☐ Rule 58.14. If an allocation job allocates a PDSE, it must specify DSNTYPE=LIBRARY.

Packaging Recommendations

- Products should supply sample jobs to perform the SMP/E APPLY and ACCEPT functions.
- Sample jobs should include clear and detailed comments. Information necessary to update the job prior to submission should be in the job, not in the Program Directory.
- If a sample job is provided on the tape, the text of the job should not appear in the Program Directory. This will reduce the size of the Program Directory, and also avoid situations where the tape and the Program Directory do not match.
- Each parameter of the DD statements in the allocation job and each parameter of the DDDEF statements in the DDDEF job should be on a separate line.
- DDDEF jobs should adhere to the following:
  1. Use ADD DDDEF, not REP DDDEF.
  2. Use the WAITFORDSN operand.
  3. Use separate job steps to divide datasets into logical groups. For example, a product could use one step for new datasets, and other steps for datasets introduced in previous releases.
- Do not specify middle-level qualifiers of VxRxMx in sample allocation jobs.
- Symbolic links for HFS or JAR files should be created in the MKDIR job, and should be relative, not absolute. In order to ensure that the MKDIR job can run multiple times without damage, products creating symbolic links in the MKDIR job should also provide UNLINK statements for every symbolic link ever created in this or previous levels, including those that have become obsolete.
- The MKDIR EXEC should be called zzzMKDIR, and the JCL invoking it should be called zzzISMKD, where zzz is the three-character prefix of the product shipping the elements.
- A PTF should not add or delete DDDEF entries, or change dataset or path names in a DDDEF entry. If this is unavoidable, the following is required:
  1. A ++HOLD ACTION is required on the PTF.
  2. The changes must be shipped in a separate DDDEF or MKDIR job shipped in the PTF, not by updating and re-shipping the existing DDDEF or MKDIR job.
  3. The new DDDEF or MKDIR job must appear in the HOLDDATA of the PTF.

End of Packaging Recommendations

For more information about packaging a RELFILE tape, see 3.1, “Relative File Tape” on page 7.

6.8 Language-Sensitive Elements

A product may have elements that require translation for national language support (NLS). In this case, you must use a base function or additive dependent function for the elements that do not have to be translated, and a separate language-support dependent function for each language into which elements are translated.

For more information, refer to Chapter 11, “Packaging for National Language Support (NLS)” on page 121.

Chapter 7. Using MCS to Define Products

SMP/E modification control statements are used to define products as function SYSMODs. This is the order of MCS for a function SYSMOD:

MCS Type	How Many
++FUNCTION	(one)
++VER	(one or more)
++IF	(none, one, or more)
++HOLD	(none, one, or more)
++MOVE	(none, one, or more)
++RENAME	(none, one, or more)
++DELETE	(none, one, or more)
++JCLIN	(none or one)
++element	(one or more to replace or update elements)

This chapter discusses the rules and MCS considerations you must follow when specifying the following statements:

- ++FUNCTION
- ++VER
- ++IF
- ++HOLD
- ++element

For more information about the ++MOVE, ++RENAME, and ++DELETE statements, see Chapter 8, "Using MCS to Manipulate Elements and Load Modules" on page 71. For more information about the ++JCLIN statement, see Chapter 9, "Using JCLIN" on page 81. For details on all MCSs, see the *SMP/E Reference* manual.

Notes:

1. Not every statement is fully documented here. The emphasis here is on those statements and operands used for packaging.
2. All input to SMP/E must be in uppercase (except comments, the LINK value on the ++HFS or the ++JAR MCS, the ALIAS value on the ++DELETE statement, and alias values in link-edit JCLIN). For details on SMP/E's syntax rules, see the *SMP/E Reference* manual.
3. All references to releases or modification levels of a function also apply to versions--each version consists of at least one release.
4. All references to PTFs also apply to SPEs--all SPEs are packaged as PTFs.

7.1 ++FUNCTION Statement

The ++FUNCTION statement identifies the SYSMOD as a base function or dependent function. A function SYSMOD may include only one ++FUNCTION statement.

Operands on the ++FUNCTION statement are used to:

- Specify the SYSMOD ID

© Copyright IBM Corp. 1986, 2003

- Identify the REWORK date
- Specify the prefix used for relative file data set names
- Specify the copyright information

7.1.1 Specifying the SYSMOD ID (sysmod_id)

The *sysmod_id* operand is the name, or SYSMOD ID, of the function. The SYSMOD ID is required, and only one value can be specified. It is also called the function modification identifier (FMID). See 10.2, “SYSMOD IDs” on page 115 for more information about the naming convention for FMIDs.

7.1.2 Identifying the REWORK Date (REWORK)

The REWORK operand indicates the date that a function was first released or last updated. The REWORK operand is important, because it can be used to distinguish every time a given function is updated and reissued with the same FMID. The date the work was done is specified as *yyyyddd*, which is the year followed by the Julian date (for example, 1993110). See 5.7, “Specifying a Rework Date” on page 34for more information.

Packaging Rules (++FUNCTION REWORK)

- ☐ Rule 59. REWORK is required on all ++FUNCTION statements, including the initial release.

  Specify the REWORK date as *yyyyddd*, which is the year followed by the Julian date (for example, 1995110).

  You must change the date every time the function is updated and reissued with the same FMID.

Note: If a SYSMOD appears more than once in the SMPPTFIN data set, the first occurrence may be received. However, none of the subsequent versions of the SYSMOD are received, even if their REWORK level is higher than the one for the first version of the SYSMOD. (Message GIM40001E is issued for each of the subsequent versions of the SYSMOD.)

7.1.3 Specifying the Prefix for RELFILE Data Sets (RFDSNPFX)

The RFDSNPFX operand identifies to SMP/E the prefix that was used in the relative file data set names for this SYSMOD. SMP/E uses this prefix when allocating data set names for the SYSMOD's relative files during RECEIVE processing. When you specify a value on the RFDSNPFX operand, remember to use that value in the names of your RELFILE data sets. For more information, see 3.1.1, “Format and Contents of the RELFILE Tape” on page 7.

Packaging Rules (Prefix for RELFILE Data Sets)

- ☐ Rule 59.1. All RELFILE data sets must start with the high-level qualifier *hlq* of your company. This qualifier must be specified on the RFDSNPFX operand of the ++FUNCTION statement, as well as on the actual RELFILE data set name. The SMPMCS file does not use the high-level qualifier; this file must be named *SMPMCS*.

  For an example of this, see Table 2 on page 8.

Packaging Rules (Prefix for RELFILE Data Sets)

☐ Rule 59.2. The DESCRIPTION operand on the ++FUNCTION statement should be, as follows:

++FUNCTION .... DESCRIPTION(*short name/descriptive name/NLV*)

where:

- "short name" is the full product/element name or its acronym, whichever is more recognizable for the product.
- "descriptive name" is BASE for the base function, and a meaningful description of the function if it is not the base function. If the product consists of only one FMID, this field may be left blank.
- "NLV" is the appropriate SMP/E three character value for national language variants (see the SMP/E Reference for a list of values). If the function contains all of the supported languages, do not use an NLV indicator.

Other rules regarding FMID DESCRIPTION:

- The FMID DESCRIPTION must specifically indicate the function contained in the FMID.
- If an FMID is shared among products, its FMID DESCRIPTION will be identical in every case.
- The description must not contain a Version, Release, or Mod Level identifier; this would cause confusion if the FMID is shared among products with different release levels.
- Use security descriptions, if appropriate, in the FMID descriptions, since features may have a mixture of security levels. For example, the feature may contain both DES and TDES FMIDs, so the description for each FMID should clearly say "DES" or "TDES".
- The description must not exceed 64 characters, including blanks.
- Use mixed case for ease of use. Use upper case for three-character NLV identifiers.
- When abbreviating, use "Srv" for Server and "Svcs" for Services.

An example of how to code the ++FUNCTION DESCRIPTION is:

```
++FUNCTION(HJE6607) REWORK(2003215)
RFDSNPFX(IBM)  FILES(2) DESCRIPTION(JES2 Base)
/****************************************************/
/* Licensed Materials - Property of IBM             */
/* This product contains "Restricted Materials of IBM"*/
/* 5647-A01 (C) COPYRIGHT IBM Corp. 1977, 1999       */
/* All Rights Reserved                              */
/* US Government Users Restricted Rights -          */
/* Use, duplication or disclosure restricted by     */
/* GSA ADP Schedule Contract with IBM Corp.         */
/****************************************************/
 .
++VER(Z038) ...
```

7.1.4 Specifying Copyright Information

The copyright date is a legal requirement.

Packaging Recommendations

Include the copyright information as a comment on the ++FUNCTION statement, after all the operands.

End of Packaging Recommendations

Here is an example of the placement of the copyright statement for a licensed program:

```
++FUNCTION(sysmod_id) FILES(nn) REWORK(yyyyddd)
/*****************************************************/
/* --- copyright statement goes here ---             */
/*****************************************************/.
```

Additional comments may be included as separate records after the copyright statement and before the final delimiter (*/).

Notes:

1. The comment statement begins in column 2.
2. If an FMID is included in more than one product, the copyright statement can refer to all the products in which that FMID is included.

7.2 ++VER Statement

The ++VER statement describes the environment required for this SYSMOD. A SYSMOD must contain a separate ++VER statement for each environment to which it applies.

The ++VER statement is required for all SYSMODs.

The operands on the ++VER statement are used to:

- Identify the SREL
- Delete SYSMODs
- Identify the base function to which a dependent function applies
- Specify mutually exclusive relationships
- Specify prerequisite relationships
- Specify requisite relationships
- Specify SYSMODs that are superseded by another SYSMOD
- Define ownership of SYSMODs

7.2.1 General Packaging Rules (++VER)

Packaging Rules (++VER)

- ☐ Rule 62. Every SYSMOD referenced on a single ++VER statement must reside in the same zone.

Packaging Rules (Multiple SYSMODs Affecting an Element)

- ☐ Rule 62.1. When two or more SYSMODs affect the same element, you must specify the relationship among those SYSMODs.
  - If both SYSMOD A and SYSMOD B ship the element (or updates to it), the MCS in both SYSMODs must define the order in which the SYSMODs should be processed (indicated by the PRE, SUP, or FMID operand) and the correct version of the element to be installed (indicated by the FMID or VERSION operand).
  - If Product A includes an element from Product B via an INCLUDE statement in a JCLIN link-edit step without changing the element, and Product A requires a particular level of Product B, then Product A's MCS must specify an unconditional requisite for the appropriate level of Product B.
  - If Product A includes an element from Product B via an INCLUDE statement in a JCLIN link-edit step without changing the element, and multiple levels of Product B would fill the needs of Product A, then Product A's program directory must identify Product B as an installation requirement, specifying the lowest acceptable level of Product B.

7.2.2 Identifying the SREL

Packaging Rules (++VER SREL)

- ☐ Rule 63. On a single ++VER statement, all SYSMODs specified on the NPRE, PRE, REQ, SUP, and VERSION operands must be applicable to the same SREL as the SYSMOD containing this ++VER statement.
- ☐ Rule 64. You must use one of the following SRELs: Z038 for z/OS, C150 for CICS, P004 for NCP, or P115 for IMS and DB2.

7.2.3 Identifying a SYSMOD's Base Function (FMID)

The FMID operand identifies the base function to which this SYSMOD applies.

Packaging Rules (++VER FMID)

- ☐ Rule 65. The FMID operand can be used only in a dependent function, not in a base function. The FMID specified in the operand must be the FMID of a base function. Both functions must be applicable to the same SREL. FMID is required for dependent functions.

Packaging Rules (++VER FMID)
☐ Rule 66. A SYSMOD cannot be both a base function and a dependent function. The FMID operand identifies a SYSMOD as a dependent function; therefore, if you specify the FMID operand, you must include it on all the ++VER statements for the SYSMOD.

7.2.4 Deleting SYSMODs (DELETE)

The DELETE operand indicates which function SYSMODs should be deleted when this function is installed. Using the DELETE operand for deleting SYSMODs is shown in Chapter 13, “SYSMOD Packaging Examples” on page 135, on pages 13.2.5, “Replacing the Initial Release” on page 139, 13.3.4, “Deleting and Superseding a Base Function” on page 143, and 13.4.5, “Deleting a Dependent Function Without Superseding It” on page 151.

DELETE is a multiple entry operand that specifies the functions to be deleted, such as a previous release of a base or dependent function.

Note: Generally, any function specified must be part of the same product. However, a new release of a product may need to delete older, equivalent releases of a different product that is applicable to the same SREL. For example, a new release of Product B might include function that was previously in Product A. In this case, Product B would need to delete all previous releases of itself and Product A. In such cases, the owner of Product B would have to negotiate with the owner of Product A for ownership approval.

The deleted function may have had requisites, JCLIN data, or other SYSMOD relationships information that must be considered when you package the deleting function. These considerations are the same as those for superseding SYSMODs, as shown in Table 8 on page 61. Table 5 on page 21 also provides more information about deleting and superseding SYSMODs.

If the specified SYSMODs are installed, SMP/E deletes them from the target and distribution libraries and from the SMP/E data sets. These SYSMODs are *explicitly deleted*. (SMP/E does not delete ++IF REQ data for SYSMODs that are explicitly deleted.

SMP/E also deletes any SYSMODs (such as PTFs) that depend on the specified SYSMODs (that is, any SYSMODs that name the specified SYSMODs on the FMID operand of their ++VER statements). These SYSMODs are *implicitly deleted*. (SMP/E does not delete ++IF REQ data for SYSMODs that are implicitly deleted.

Notes:

1. If a function requires any service that was previously installed on a deleted function, the user may have to reinstall that service. (This may be the case when a PTF applies to more than one release of a function.) When SMP/E installs the deleting SYSMOD, it will identify which SYSMODs are being deleted.
2. Starting with SMP/E Release 7, SMP/E tracks when a module is deleted from a load module composed of modules to be deleted and modules not to be deleted. For each deleted module, SMP/E keeps a record of the connection

between the deleted module and the load module. If any of these deleted modules are ever reintroduced, SMP/E looks for load modules having a record of a connection to those modules, and automatically rebuilds the load modules to include these modules again.

If you are replacing a product that contained cross-product modules or load modules, and the new release of the product eliminates the previous cross-product connections without deleting the modules or load modules that were involved, you need to ensure through packaging of the new release that SMP/E does not try to perpetuate the previous cross-product connections. For examples, see 9.6.4.1, "Linking a Module from Another Function" on page 100 and 9.6.4.2, "Linking Modules into a Load Module for Another Function" on page 101.

Packaging Rules (++VER DELETE)

- ☐ Rule 67. If the DELETE operand is used in a function, it must only specify the FMID of a base function or a dependent function.
- ☐ Rule 68. Every product (other than the initial release) must use ++VER DELETE to delete all of its previous releases and versions. Each base function must delete all related previous base functions, and each dependent function must delete all related previous dependent functions.

  For example, Release 2's primary base function must delete Release 1's primary base function, and Release 2's Japanese NLV dependent function must delete Release 1's Japanese NLV dependent function.

  Refer to 4.2.3.4, "Deleting and Superseding SYSMODs" on page 20 for detailed information about superseding and deleting previous releases.

  Note: Optionally, dependent functions can delete previous releases and versions of the product.
- ☐ Rule 69. A language-support dependent function must not delete an additive dependent function, and vice versa.
- ☐ Rule 70. A function cannot delete itself.
- ☐ Rule 71. For a service update, the DELETE operand must include all the FMIDs that were specified on the DELETE operand in the original function SYSMOD.

For more information about deleting SYSMODs, refer to 4.2.3.4, "Deleting and Superseding SYSMODs" on page 20.

Packaging Recommendations

- You should specify additive dependent functions that are applicable to a deleted base function. This allows customers to determine what is deleted by a function by reading the associated MCS. (Specifying these functions is for documentation purposes only. Dependent functions are automatically deleted when the associated base functions are deleted.)
- It is not necessary to specify language-support dependent functions that are applicable to a deleted base function. These functions are automatically deleted when the associated base functions are deleted.

- To improve SMP/E performance during installation, very large products should consider providing users with an example of how to package the ++VER DELETE information separately in a dummy function SYSMOD.

  This dummy function SYSMOD is received, applied, and accepted to delete the previous releases of your product from the existing target and distribution libraries, and UCLIN is run to delete the SYSMOD entries for the deleted function and for the dummy function. The new release of the product is then installed.

  For example, assume the previous release of your product is MYFUNC1, and you want to explain to users how to delete it with dummy function DELFUNC. MYFUNC1 is applicable to SREL Z038 and is installed in target zone TGT1 and distribution zone DLIB1. Here is an example of the dummy function, along with the instructions you should provide to your users:

```
++FUNCTION(DELFUNC)     /* Any valid unique SYSMOD ID.   */.
++VER(Z038)             /* For SREL Z038 (z/OS products). */
     DELETE(MYFUNC1)    /* Deletes MYFUNC1.              */.
```

  These are the commands you use to receive and install the dummy function, and to delete the SYSMOD entries for the dummy function and the deleted function:

```
SET     BDY(GLOBAL)      /* Set to global zone.       */.
RECEIVE S(DELFUNC)       /* Receive the function.     */.
SET     BDY(TGT1)        /* Set to applicable target. */.
APPLY   S(DELFUNC)       /* Apply to delete old       */
                         /* function.                 */.
SET     BDY(DLIB1)       /* Set to applicable DLIB.   */.
ACCEPT  S(DELFUNC)       /* Accept to delete old      */
                         /* function.                 */.
SET     BDY(TGT1)        /* Set to applicable target. */.
UCLIN.
DEL     SYSMOD(DELFUNC)  /* Delete SYSMOD entries for */.
DEL     SYSMOD(MYFUNC1)  /* dummy and old function.   */.
ENDUCL.
SET     BDY(DLIB1)       /* Set to applicable DLIB.   */.
UCLIN.
DEL     SYSMOD(DELFUNC)  /* Delete SYSMOD entries for */.
DEL     SYSMOD(MYFUNC1)  /* dummy and old function.   */.
ENDUCL                   /*                           */.
```

  When you accept the dummy function, SMP/E automatically deletes the associated SYSMOD entry from the global zone and the MCS entry from the SMPPTS.

  To complete the cleanup, you should also use the REJECT command to delete any SYSMODs and HOLDDATA applicable to the dummy function and the old function. In addition, you should delete the FMIDs from the GLOBALZONE entry to prevent SMP/E from receiving any SYSMODs or HOLDDATA applicable to either of those functions. Here are examples of the commands you can use to do this.

```
SET     BDY(GLOBAL)      /* Set to global zone.       */.
REJECT HOLDDATA NOFMID   /* Reject SYSMODs, HOLDDATA  */
       DELETEFMID        /* for the deleted functions.*/
       (DELFUNC MYFUNC1) /* Delete the FMIDs from the */
                         /* GLOBALZONE entry.         */.
```

End of Packaging Recommendations

7.2.5 Specifying Mutually Exclusive SYSMODs (NPRE)

The NPRE operand is an optional, multiple-entry operand. It indicates which function SYSMODs are mutually exclusive and cannot exist in the same zone as the specifying function. The SYSMOD ID specified on the NPRE operand cannot be already installed and must not be superseded by a SYSMOD being installed concurrently. These are called negative prerequisite SYSMODs. Using the NPRE operand for mutually exclusive functions is shown under Chapter 13, “SYSMOD Packaging Examples” on page 135 on page 13.6, “Example 5: Mutually Exclusive Dependent Functions” on page 155. Also see 4.2.3.3, “Negative Prerequisite SYSMODs” on page 20 for more information.

Packaging Rules (Mutually Exclusive Versions)

- ☐ Rule 72. If the NPRE operand is used in a base function, it can only specify the FMID of a base function. If the NPRE operand is used in a dependent function, it can specify the FMID of a base function, the FMID of a dependent function, or both. In either case, all functions involved must be applicable to the same SREL.

7.2.6 Specifying Prerequisite Relationships (PRE)

The PRE operand is an optional, multiple-entry operand for dependent functions. It indicates which SYSMODs are prerequisites for the specifying SYSMOD. A prerequisite must either be already installed, or must be installed concurrently with the specifying SYSMOD. Using the PRE operand for prerequisite SYSMODs is shown under Chapter 13, “SYSMOD Packaging Examples” on page 135 on pages 13.2.3, “PTF Service That Depends on Previous Service” on page 137, 13.4.6, “Establishing the Order of Additional Dependent Functions” on page 151, and 13.5.1, “Initial Release of a Base Function with a Functional Prerequisite” on page 152. See 4.2.3.1, “Prerequisite SYSMODs” on page 19for more explanation.

Note: You cannot use PRE to assume ownership of an element from another function. A dependent function must use the FMID operand, and a base function must use the VERSION operand on the ++*element* statement.

Packaging Rules (++VER PRE)

- ☐ Rule 74. The PRE operand can be used in a base or dependent function. It can specify the FMID of a base function or a dependent function, or it can specify a PTF number. In any case, all functions involved must be applicable to the same SREL.

  Note: Do not use the PRE operand in a dependent function to indicate its own base function. You must use the FMID operand for this purpose.

- ☐ Rule 75. The specified prerequisite (or a valid replacement) must be available as long as the specifying SYSMOD is available. When neither the prerequisite function nor the replacement SYSMOD is available, all the functions specifying the prerequisite must be repackaged.

Packaging Rules (++VER PRE)

- ☐ Rule 76. If a dependent function specifies a PTF as a prerequisite, the dependent function and the PTF must be applicable to the same base function.

7.2.7 Superseding SYSMODS (SUP)

The SUP operand is an optional, multiple-entry operand. It indicates which SYSMODs are contained in and replaced by this SYSMOD. For example, it could be used for a new release of a dependent function or for a service update. When a SYSMOD specifies SUP on its ++VER statement, this indicates to SMP/E that the superseded SYSMODs do not need to be installed once the superseding SYSMOD has been installed. Using the SUP operand for superseding SYSMODs is shown under Chapter 13, “SYSMOD Packaging Examples” on page 135 on pages 13.2.2, “PTF Service for the Initial Release” on page 137, and 13.3.4, “Deleting and Superseding a Base Function” on page 143. Table 5 on page 21 also provides a comparison of deleting and superseding SYSMODs.

Note: You cannot use SUP to assume ownership of an element from another function. A dependent function must use the FMID operand, and a base function must use the VERSION operand on the ++*element* statement.

Packaging Rules (++VER SUP)

- ☐ Rule 77. If the SUP operand is used in a base function, it can specify the FMID of a base function, the FMID of a dependent function, a PTF number, or an APAR number. If the SUP operand is used in a dependent function, it can specify the FMID of a dependent function, a PTF number, or an APAR number. In either case, all functions involved must be applicable to the same SREL.
- ☐ Rule 78. A function must provide all the supported function contained in all the SYSMODs it supersedes.
- ☐ Rule 79. All the superseded SYSMODs must be in the same product as the superseding SYSMOD.
- ☐ Rule 80. For each environment (++VER FMID and SREL), all the elements in the superseded SYSMODs must be contained either in the superseding SYSMOD or in the combination of the superseding SYSMOD and its requisites
 (other SYSMODs specified on the ++VER REQ or PRE operands, or on a ++IFREQ statement)
 unless the element is deleted by the superseding SYSMOD.

Packaging Rules (++VER SUP)

- ☐ Rule 81. The environment of a superseded SYSMOD must not be at a higher functional level than the level of the superseding function.
  - If the superseded SYSMOD is a base function, it must apply to the same SREL as the superseding SYSMOD.
  - If the superseded SYSMOD is a dependent function, it must apply to the same SREL as the superseding SYSMOD. In addition, the superseded dependent function must do one of the following:
    - Be applicable to the same base function as the superseding dependent function
    - Be applicable to a lower-level function than the superseding function
- ☐ Rule 82. A new release of a base function can supersede a previous release of that base function only if it also deletes that previous release. Likewise, a new release of a base function can supersede a dependent function applicable to a previous release of that base function only if the new release also deletes that dependent function.
- ☐ Rule 83. A new dependent function can supersede previous releases of that dependent function only if it also deletes those releases.
- ☐ Rule 84. A service update must supersede any PTFs and APAR fixes that are incorporated into it.
- ☐ Rule 85. A superseding function (or its requisites) must carry on the SYSMOD relationships defined in the superseded function SYSMODs. Table 8 on page 61 shows the relationships and processing information that the superseding SYSMOD or its requisites may need to include from the superseded SYSMODs.

  Note: Table 8 on page 61 also applies to deleting SYSMODs and the information that they or their requisites may need to include from the deleted SYSMODs.

Packaging Recommendations

- A new release of a function should both delete and supersede the previous release if all of the following are true:
  - The new release contains at least all the function that was in the previous release.
  - If other products specified the deleted function as a requisite, all the internal and external interfaces used by those other products are unchanged in the new release.
  - Other products that specified the previous release as a requisite can run with the new release.
- Evaluate a replacement function using Table 5 on page 21 as a guide. If the replacement function matches that description, then the preferred and recommended way to replace the previous function is to both delete and supersede it.

End of Packaging Recommendations

Table 8 (Page 1 of 2). Considerations for Superseding (and Deleting) SYSMODs

If the superseded (and deleted) SYSMOD(1) specified this:	**Evaluate whether the statement is still valid and do the following as appropriate:**
++VER PRE(*sysmod*,...)	Specify ++VER PRE or SUP for the same SYSMODs *(or)* Specify ++VER PRE or REQ for another SYSMOD(3) that is either superior to or that specifies ++VER PRE or SUP for the same SYSMODs
++VER REQ(*sysmod*,...)	Specify ++VER SUP, PRE, or REQ for the same SYSMODs *(or)* Specify ++VER PRE or REQ for another SYSMOD(3) that is either superior to or that specifies ++VER SUP, PRE, or REQ for the same SYSMODs
++VER SUP(*sysmod*,...)	Specify ++VER SUP for the same SYSMODs
++VER VERSION(*sysmod*,...) or ++element VERSION(*sysmod*,...)	Specify ++VER VERSION or ++element VERSION for the same SYSMODs **Note:** The ++VER VERSION value affects all new or replacement elements that do not specify an overriding ++element VERSION value.
++IF FMID(*fmid*) REQ(*sysmod*,...)	Specify ++IF REQ for the same SYSMODs *(or)* Specify ++IF REQ for another SYSMOD(3) that is either superior to or that specifies ++VER SUP, PRE, or REQ for the same SYSMODs *(or)* Specify ++VER PRE or REQ for another SYSMOD(3) that is either superior to or that specifies ++IF REQ for the same SYSMODs **Note:** All of the ++IF statements must specify the same FMID value as the original ++IF statement.
++HOLD statement	Evaluate to see whether the ++HOLD statement is required, or can be deleted by updating the installation documentation
++MOVE statement	Include the ++MOVE statement, unless: • An element statement deletes the element • A ++DELETE statement deletes the load module. **Note:** When moving an element, make sure to specify the new libraries on the DISTLIB and SYSLIB operands in the appropriate ++element statements.
++RENAME statement	Include the ++RENAME statement, unless a ++DELETE statement deletes the load module.
++DELETE statement	Include the ++DELETE statement.
++JCLIN statement and JCLIN data	Include the ++JCLIN statement and JCLIN data, merging the ++JCLIN operands and JCLIN data from SYSMOD(1). **Note:** If several SYSMODs are superseded (or deleted), merge the JCLIN data so that the most recent data is properly reflected in SYSMOD(2).
++MOD CSECT(*name*)	Include the CSECT data.
++MOD LMOD(*name*)	Evaluate to see whether the LMOD operand is still required on the ++MOD statement. **Note:** The new JCLIN data may eliminate the need for the LMOD operand.

Table 8 (Page 2 of 2). Considerations for Superseding (and Deleting) SYSMODs	
If the superseded (and deleted) SYSMOD(1) specified this:	**Evaluate whether the statement is still valid and do the following as appropriate:**
Element updates	Merge all of the updates contained in the superseded (or deleted) SYSMODs into the new elements
UCLIN data	Evaluate to see whether the UCLIN data is required, or whether an alternative to UCLIN may be used
UCLIN to move an element or load module	Use a ++MOVE statement to move the element or load module, unless: • An element statement deletes the element • A ++DELETE statement deletes the load module.
UCLIN to rename a load module	Use a ++RENAME statement to rename the load module, unless a ++DELETE statement deletes the load module.
UCLIN to delete a load module	Use a ++DELETE statement to delete the load module.

Packaging Rules (Moving and Replacing Elements)

☐ Rule 86. The ++VER statement for each SYSMOD that contains an element that is replaced or moved to a new library must use the PRE or SUP operand to specify the previous SYSMOD, if any, that also replaced or moved that element.

7.2.8 Defining Ownership (VERSION)

The VERSION operand specifies one or more dependent function SYSMODs whose elements should be considered functionally lower than the version of those elements in the specifying function SYSMOD. The VERSION operand is also used to add a version of an element to a dependent function when that element exists only in lower-level dependent functions.

When a dependent function SYSMOD that specifies the VERSION operand on the ++VER or element statement is installed, the dependent function will assume ownership of the elements from the functions specified on the VERSION operand. Subsequent processing of service SYSMODs or USERMODs applicable to the functions that previously owned the elements will not update or replace the affected elements.

Packaging Rules (++VER VERSION)

☐ Rule 87. You must specify the lower-level function SYSMODs on the VERSION operand of each ++VER statement in the higher-level function SYSMOD.

VERSION is required to establish which elements are functionally higher when SYSMODs for different dependent functions have elements with the same name and type in common. Also, specifying the lower-level function SYSMODs on the VERSION operand on the ++VER statement in the higher-level function SYSMODs ensure that ownership of the elements is given to the highest level SYSMOD.

Packaging Rules (++VER VERSION)

- ☐ Rule 88. If a dependent function uses the VERSION operand, any subsequent function replacing this dependent function must contain all the elements whose ownership was assumed by the dependent function.
- ☐ Rule 90. A new release of a dependent function can have elements in common with a lower-level dependent function for the same base function. If so, the new release must incorporate those elements and, if the lower-level dependent function is not deleted, must establish the superiority of its version of those elements, as well as its installation relationship with the lower-level function. The superiority of the elements is established by the VERSION operand on either the ++VER or element statement. The installation relationship is established by either the PRE or SUP operand on the ++VER statement. For more information, see the descriptions of these operands elsewhere in this chapter.
- ☐ Rule 91. VERSION must specify all the dependent functions that are functionally lower than the specifying function and that include the elements to be versioned.
- ☐ Rule 92. The VERSION operand must be specified on the ++VER statement if all elements affected by this SYSMOD are to be versioned the same way. The VERSION operand must be specified on the element statement if individual elements can be versioned differently.

Notes:

1. For the VERSION operand to take effect, the specified functions must be installed in the same zone as the specifying SYSMOD.
2. If a PTF creates a new element version for a given function, it must use the VERSION operand on the ++VER statement. For more information, see the Process Documentation References (PDRs).

Packaging Recommendations

If use of the VERSION operand between two products is unavoidable, it is the responsibility of the development owner of Product B to ensure that the development owner of Product A understands and agrees to what has been done.

VERSION can also be specified on an element statement to establish the functional level of elements and override the VERSION values specified on the ++VER statement. However, the VERSION operand on the element statement is not additive; it does not automatically take over ownership from the functions specified on the ++VER VERSION operand. To take over ownership from any of the functions specified on the ++VER VERSION operand, you must repeat those values on the VERSION operand for the element statement.

End of Packaging Recommendations

7.3 ++IF Statement

The ++IF statement defines conditional requisites. This is an optional statement associated with the ++VER statement that precedes it. Several ++IF statements may be associated with a single ++VER statement. If a SYSMOD contains several ++VER statements, there may be ++IF statements associated with each one. Using the ++IF statement is shown under Chapter 13, “SYSMOD Packaging Examples” on page 135 on pages 13.2.4, “Ensuring That a Fix for a Previous Release Is Not Lost” on page 138, 13.3.3, “Cross-Product Service between Corequisite Base Functions” on page 142, 13.4.3, “Corequisite PTFs with an Element Common to the Base and Dependent Functions” on page 145, 13.4.4, “Corequisite PTFs with All Elements Common to Base and Dependent Functions” on page 148, 13.4.7, “Conditional Corequisite Dependent Functions” on page 152, 13.5.2, “Dependency on an SPE or Service for Another Base Function” on page 153, and 13.5.3, “Cross-Product Service for a Base Function with a Prerequisite” on page 154.

The operands of the ++IF statement are used to:

- Specify the function to which the condition applies
- Specify the SYSMODs that must be installed if the condition exists

7.3.1 Specifying the Function to which the Condition Applies (FMID)

The FMID operand is a required, single-entry operand. It indicates the function to which the conditional requisite applies.

Packaging Rules (++IF FMID)

- ☐ Rule 92.1. The ++IF statement can be used in a base function or a dependent function. In both cases, the FMID operand can specify either a base function or a dependent function.
- ☐ Rule 93. The function cannot specify its own FMID.

  Note: This Rule does not apply to products that require installation using the OS/390 Release 3 (or later) level of SMP/E.

- ☐ Rule 95. If the FMID operand is used in a base function, the specified SYSMOD must be in a previous *version* of the product.

  For example, Version 2 Release 2 of a product cannot specify ++IF FMID for Version 2 Release 1; however, it can specify Version 1 Release 3.

Notes:

1. A service update of a base function can specify a function SYSMOD that is in the same product version when the ++IF statement comes from a concurrently-installed PTF. or APAR fix that was integrated into the base function.
2. A dependent function can specify any function SYSMOD, regardless of whether two functions are part of the same product or product version.

Packaging Recommendations

- The ++IF MCS should include a comment to identify the product required by the FMID operand.
- Provide ++IF REQs for all functionally required service, with comments explaining the reason for the REQ.

End of Packaging Recommendations

7.3.2 Specifying Requisite Conditions (REQ)

The REQ operand is a required, multiple-entry operand. It specifies one or more SYSMODs that must be installed if the function SYSMOD specified on the FMID operand of the ++IF statement is installed.

- If the specified function is already installed (or is currently being installed) in the same zone where the specifying SYSMOD is being installed, the requisite must also be installed in that zone; otherwise, the specifying SYSMOD will not be installed.
- If the specified function is not yet installed in the zone, SMP/E saves the information from the ++IF statement in case the specified function is installed later.

In both cases, SMP/E saves requisite data from the ++IF statement, even if the function specified on the ++IF FMID operand is restored or deleted.

Note: The specified SYSMOD may be in the same or a different product or product version as the specifying SYSMOD.

Packaging Rules (++IF REQ)

- ☐ Rule 96. The REQ operand can be used in a base function or a dependent function. In both cases, the REQ operand can specify either a dependent function or a PTF number.
- ☐ Rule 97. Any dependent function specified on the REQ operand (or a valid replacement) must be announced and must be available as long as the specifying SYSMOD is available.
- ☐ Rule 98. If the specified conditional requisite is a function and it is deleted by a new release of that function, one of the following must be done:
  - The new release can also supersede the specified requisite function. This way, the function specifying the requisite does not need to be repackaged.
  - If the specified requisite function is to be deleted by a new release without also being superseded, the specifying function must be repackaged and redesigned to refer to the new release as the requisite.
- ☐ Rule 99. If the specified conditional requisite is a PTF, any subsequent replacement must supersede the specified PTF. This eliminates the need to repackage the specifying function to redefine the conditional requisite.

Packaging Rules (++IF REQ)

- ☐ Rule 100. A SYSMOD cannot specify both a conditional and unconditional relationship for the same SYSMOD ID.

  For example, the following statements cannot appear in the same SYSMOD:

  ```
  ++VER REQ(ABC1234).
  ++IF FMID(Z) REQ(ABC1234).
  ```

  Note: This Rule does not apply to products that require installation using the OS/390 Release 3 (or later) level of SMP/E.

- ☐ Rule 100.1. If the specified SYSMOD is a dependent function, the FMID to which it applies must either:

  – Match the FMID specified on the associated ++IF statement contained in the specifying SYSMOD

  – Unconditionally coexist with the FMID specified on the associated ++IF statement contained in the specifying SYSMOD

7.4 ++element Statement

Element statements describe the elements contained in a SYSMOD and are used by SMP/E to select which elements should be installed in the target and distribution libraries. If an element statement is not provided for an element, the element is not installed, even if it was defined in the JCLIN data. The following statements can be used to add or replace elements:

- ++MAC describes a new or replacement macro.
- ++MOD describes a new or replacement module.
- ++SRC describes new or replacement source code.
- ++HFS and ++JAR describes new or replacement elements that are installed in a hierarchical file system (HFS) or Java Archive (JAR) file.
- Data element MCSs describe new or replacement elements that are not macros, modules, or source code. Types of data elements are shown in Table 7 on page 39 under 6.4, "Data Element Types" on page 39.

A single SYSMOD can contain any one of the following statements or combinations of statements for a given macro, module, or source element name.

- ++**MAC statement:** Macros may be used during source (++SRC) assemblies and can be used to assemble source that is not defined by ++SRC statements. The resulting object modules are written to a work data set that is used as SYSPUNCH input to link-edit the modules into the target libraries. If you package code that is to be processed this way, you must provide for the JCLIN data that defines the assembly and link-edit steps to SMP/E. (This JCLIN data may be packaged with the code or created during a generation procedure.)
- ++**MOD statement:** A module can be link-edited into a load module in a target library, or, for a single-module load module, can be copied into a target library. If you package code that is to be processed this way, you must provide for the

JCLIN data that defines the link-edit or copy steps to SMP/E. (This JCLIN data may be packaged with the code or created during a generation procedure.)

- ++**SRC statement:** Source can be supplied without the corresponding modules to cause the source to be assembled. The resulting object modules are written to a work data set that is used as SYSPUNCH input to link-edit the modules into the target libraries. If you package code that is to be processed this way, you must provide for the JCLIN data that defines the assembly and link-edit steps to SMP/E. (This JCLIN data may be packaged with the code or created during a generation procedure.)
- ++**SRC and** ++**MOD statements:** A module may be provided in both source and executable forms. (Each form represents a different element type, and both must be in the same FMID.) In this case, the source will not be assembled. Users who do not need to change the source code will have an executable module they can install. Users who do need to change the source code can make those changes to the source so that it will be assembled to create an object module. The object module is installed as described above for the ++MOD statement.

  If you package a module that is to be processed this way, you must provide for the JCLIN data that defines the link-edit or copy steps to SMP/E. (This JCLIN data may be packaged with the code or created during a generation procedure.)

Packaging Recommendations

If you package an element with a ++SRC statement, you should also include the associated ++MOD statement.

End of Packaging Recommendations

Packaging Recommendations

Do not use ++MAC, ++MOD, or ++SRC statements to package elements that are not macros, modules, or source, respectively. Use data element statements or HFS or JAR element statements (as appropriate) to package such elements.

End of Packaging Recommendations

These are the element statement operands used to package SYSMODs:

Table 9. Operands for Element Statements

Operands	Macros	Modules	Source	HFS or JAR Elements (SMP/E 3.2 and Later Releases Only)	Data Elements (SMP/E R5 and Later Releases Only)
RELFILE	++MAC	++MOD	++SRC	++HFS or ++JAR	All data element MCS
ALIAS (SMP/E R5 and later releases only)					All data element MCS
BINARY (SMP/E 3.2 and later releases only)				++HFS or ++JAR	
CSECT		++MOD			
DALIAS		++MOD			
DELETE	++MAC	++MOD	++SRC	++HFS or ++JAR	All data element MCS
DISTLIB	++MAC	++MOD	++SRC	++HFS	All data element MCS
LINK (SMP/E 3.2 and later releases only)				++HFS or ++JAR	
LMOD		++MOD			
MALIAS	++MAC				
PARM (SMP/E 3.2 and later releases)				++HFS or ++JAR	
RMID (service updates only)	++MAC	++MOD	++SRC	++HFS or ++JAR	All data element MCS
SYSLIB	++MAC		++SRC	++HFS or ++JAR	All data element MCS
TALIAS		++MOD			
TEXT (SMP/E 3.2 and later releases only)				++HFS or ++JAR	
UMID (service updates only)	++MAC	++MOD	++SRC		
VERSION	++MAC	++MOD	++SRC	++HFS or ++JAR	All data element MCS

Packaging Rules (DISTLIB for Elements)

- □ Rule 101. Do not use SYSPUNCH as the DISTLIB. It is used by SMP/E and other products to process assembled modules.
- □ Rule 101.1. Do not specify a pathname in a hierarchical file system (HFS) or java archive file (JAR) as the DISTLIB.

Packaging Rules (DISTLIB for Elements)
☐ Rule 101.1a. Do not specify SMP/E temporary data sets (SMPLTS, SMPMTS, SMPPTS, SMPSTS, etc.) as DISTLIB or SYSLIB values on MCS.

For details on specifying these operands, see the *SMP/E Reference* manual.

Chapter 8. Using MCS to Manipulate Elements and Load Modules

Modification Control Statements can help you address packaging goals that at one time could be done only through UCLIN. This chapter describes how you can use MCS to:

- Move macros, modules, source, and load modules (++MOVE statement)
- Rename load modules (++RENAME statement)
- Delete load modules (++DELETE)
- Delete elements (++element DELETE)
- Enable load module changes at the CSECT level (++MOD CSECT)
- Change ownership of elements

Note: Regardless of the order in which ++MOVE, ++RENAME, and ++DELETE statements are coded in a SYSMOD, they are always processed in this order:

- ++MOVE
- ++RENAME
- ++DELETE

Afterwards, ++JCLIN statements are processed, followed by element statements.

Table 10 summarizes how you can use MCS to manipulate elements and load modules.

Table 10 (Page 1 of 2). Performing Actions on Elements and Load Modules

Goal	Packaging Solution	Where to Find More Information
Add a module to a load module.	To add a module to an existing load module, use the LMOD operand on the ++MOD statement. To add a module and create a new load module, use JCLIN data.	See 9.2, “When Do You Need JCLIN?” on page 82.
Change the owner of an element.	If a new function deletes an old function, the DELETE operand on the ++VER statement indicates that the owner has changed.	See 7.2.4, “Deleting SYSMODs (DELETE)” on page 55.
	If a new dependent function introduces a higher-level version of the element, you can use the VERSION operand on the ++VER or element statement to indicate that the owner has changed.	See 7.2.8, “Defining Ownership (VERSION)” on page 62.
	If an element is being migrated from one base function to another, you can use the VERSION operand on the element statement to indicate that the owner has changed.	For more information on migration to a new function, see 13.8.3, “Migrating Elements by Updating Both Functions” on page 162. For more information on migration using a PTF, see 13.8.4, “Migrating Elements by Using a PTF” on page 163.

© Copyright IBM Corp. 1986, 2003

Table 10 (Page 2 of 2). Performing Actions on Elements and Load Modules		
Goal	**Packaging Solution**	**Where to Find More Information**
Move a macro, module, source, or load module.	Use the ++MOVE statement.	See 8.1, "Moving Elements and Load Modules (++MOVE)" on page 72.
Rename a load module.	Use the ++RENAME statement.	See 8.2, "Renaming Load Modules (++RENAME)" on page 74.
Delete a load module.	Use the ++DELETE statement.	See 8.3, "Deleting Load Modules (++DELETE)" on page 75.
Delete a module (CSECT) from a load module.	Use the ++MOD CSECT operand	See 8.5, "Enabling Load Module Changes at the CSECT Level (++MOD CSECT)" on page 77.

8.1 Moving Elements and Load Modules (++MOVE)

The ++MOVE statement moves a macro, module, source, or load module from its current library to another library.

Packaging Restriction

Restriction 15.2. The ++MOVE statement is not allowed for data elements, HFS or JAR elements, or ++PROGRAM elements. This is an SMP/E restriction.

End of Packaging Restriction

This is an optional statement. If you include it, it must immediately follow the last ++HOLD statement, or if there are none, it must follow the last ++VER statement or the last ++IF statement associated with that ++VER statement. It must precede all other MCS (++RENAME, ++DELETE, ++JCLIN, and element statements).

Packaging Rules (++MOVE)

- ☐ Rule 101.2. If an element needs to be moved, a ++MOVE statement must be used instead of UCLIN.
- ☐ Rule 102. A dependent function can contain a ++MOVE statement for an element or load module it does not contain only if the element or load module is owned by the base function to which the dependent function applies, or by another dependent function for the same base function. In either case, the moving dependent function must specify the owning function as a prerequisite.

  If a previous dependent function has performed a ++MOVE on the element or load module, then the new dependent function must specify that dependent function as a prerequisite.

Packaging Rules (++MOVE)

- ☐ Rule 103. A function can contain only one ++MOVE statement for a given element.
- ☐ Rule 104. A function can contain no more than two ++MOVE statements for a given load module, one for each SYSLIB defined for the load module.
- ☐ Rule 105. All MCS following the ++MOVE statements and referring to the elements or load modules that were moved must reflect the new libraries for those elements or load modules. All SYSMODs applied subsequent to the move must reflect the new libraries for those elements or load modules.
- ☐ Rule 106. All changes caused by a ++MOVE MCS must also be specified in any JCLIN and SYSGEN macros that refer to the moved member.
- ☐ Rule 107. If SYSMOD(1) defines or moves an element, subsequent SYSMODs containing that element must specify SYSMOD(1) as a prerequisite.
- ☐ Rule 108. If SYSMOD(1) moves a given load module using a ++MOVE statement, any SYSMOD that supersedes SYSMOD1 must also contain the ++MOVE statement.
- ☐ Rule 109. If an element or load module to be moved to a new SYSLIB is a member of a totally copied library, the moving function must also move the same element or corresponding module to a new distribution library.

Packaging Recommendations

- A base function should not contain a ++MOVE statement, unless a PTF containing the statement was integrated into a service update of that function.
- New releases of a base function do not own elements that would need to be moved from one library to another. However, there may be shared load modules that should be moved. In this instance, a base function may contain a ++MOVE for the shared load module.

End of Packaging Recommendations

Two ++MOVE statements are allowed for a load module because load modules can exist in two target libraries.

SMP/E processes the ++MOVE statements in a SYSMOD first; therefore, any MCS after the ++MOVE need to reflect the element's new library.

You must ensure that a totally copied library structure (a DISTLIB that was totally copied to a target library) is defined to SMP/E. If a load module was moved from the defined target library, its corresponding module must be moved to a new distribution library. This ensures that the totally copied structure is defined to SMP/E.

8.2 Renaming Load Modules (++RENAME)

The ++RENAME statement changes the name of a load module.

This is an optional statement. If you include it, it must immediately follow the last ++HOLD or ++MOVE statement, or if there are none, it must follow the last ++VER statement or the last ++IF statement associated with that ++VER statement. It must precede all other MCS (++DELETE, ++JCLIN, and element statements).

All MCS that follow the ++RENAME statements that refer to the load modules that were renamed must reflect the new name for those load modules.

SMP/E will not rename any aliases associated with the specified load module.

Packaging Rules (++RENAME)

- ☐ Rule 110. A dependent function can contain a ++RENAME statement for a load module associated with either the base function to which it applies, or with another dependent function that is applicable to that same base function and that is required by the function containing the ++RENAME statement.
- ☐ Rule 111. All changes caused by a ++RENAME MCS must also be specified in any JCLIN and SYSGEN macros that refer to the old name of the load module.
- ☐ Rule 112. A function can contain only one ++RENAME statement for a given load module.
- ☐ Rule 113. If SYSMOD(1) renames a given load module using a ++RENAME statement and SYSMOD(2) defines that load module under its new name with JCLIN data, SYSMOD(2) must specify its relationship to SYSMOD(1) using the PRE, DELETE, or SUP and DELETE operands on its ++VER statement.
- ☐ Rule 114. If SYSMOD(1) defines a given load module and SYSMOD(2) renames that load module using a ++RENAME statement, SYSMOD(2) must specify its relationship to SYSMOD(1) using the PRE operand on its ++VER statement.
- ☐ Rule 115. If a load module being renamed was totally copied from a distribution library into a target library (defined by JCLIN data as a totally copied load module), this function must also use a ++MOVE statement to move the identically named element (++MOD) to a new distribution library.
- ☐ Rule 116. If a dependent function is renaming a load module, that function must refer to the last previous lower-level dependent function (if any) that (1) moved the load module being renamed or (2) renamed a load module to the name of the load module being renamed again.
  - If that previous dependent function moved the load module being renamed, this dependent function can either delete or supersede and delete that dependent function, or specify it as a prerequisite.
  - If the previous dependent function renamed a load module to the name of the load module being renamed again, this dependent function must specify that previous dependent function as a prerequisite.

Packaging Recommendations

- A base function should not contain a ++RENAME statement, unless a PTF containing the statement was integrated into a service update of that function.

  New releases of a base function do not own elements that would need to be renamed. However, there may be shared load modules that should be renamed. In this instance, a base function may contain a ++RENAME for the shared load module.

- If you want to rename a load module and use inline JCLIN to create a new load module with the original name of the renamed load module, you must package your changes in two SYSMODs: one to rename the existing load module, and one to create the new load module.

  The two SYSMODs must not state any relationship to each other and must be applied separately: first the SYSMOD that renames the existing load module, then the one that creates the new load module.

  If the SYSMODs need to be restored, they must also be restored separately, in the reverse order of the installation: first the SYSMOD that created the new load module, then the one that renamed the existing load module.

End of Packaging Recommendations

8.3 Deleting Load Modules (++DELETE)

The ++DELETE statement deletes a load module and any of its aliases from its current target library. It can also delete the aliases without deleting the load module.

This is an optional statement. If you include it, it must immediately follow the last ++HOLD, ++MOVE, or ++RENAME statement, or if there are none, it must be associated with that ++VER statement or the last ++IF statement associated with that ++VER statement. It must precede all other MCS (++JCLIN and element statements).

Using the ALIAS operand deletes the alias of a load module without deleting the load module itself. If the ALIAS operand is not specified, the load module and all of its aliases are deleted.

Packaging Rules (++DELETE)

- ☐ Rule 117. A dependent function can contain a ++DELETE statement for a load module associated with either the base function to which it applies, or with another dependent function that is applicable to that same base function and that is required by the function containing the ++DELETE statement.
- ☐ Rule 118. A function can contain only one ++DELETE statement for a given load module.
- ☐ Rule 119. A function containing a ++DELETE statement must also include the appropriate changes for its JCLIN or SYSGEN macros (if any) to reflect the change.

Packaging Rules (++DELETE)

- Rule 120. If SYSMOD(1) deletes a given load module using a ++DELETE statement and SYSMOD(2) defines that load module with JCLIN data, SYSMOD(2) must specify its relationship to SYSMOD(1) using the PRE, DELETE, or SUP and DELETE operands on its ++VER statement.
- Rule 121. If SYSMOD(1) defines a given load module with JCLIN data and SYSMOD(2) deletes that load module using a ++DELETE statement, SYSMOD(2) must specify its relationship to SYSMOD(1) using the PRE or FMID operand on its ++VER statement.
- Rule 122. A dependent function that is deleting a load module must refer to the last previous lower-level dependent function (if any) that (1) moved the load module being deleted or (2) renamed a load module to the name of the load module being deleted.
  - If that previous dependent function moved the load module being deleted, this dependent function can either delete or supersede and delete that dependent function or specify it as a prerequisite.
  - If that previous dependent function renamed a load module to the name of the load module being deleted, this dependent function can either delete or supersede and delete that dependent function or specify it as a prerequisite.
- Rule 122.1. If a SYSMOD is deleting an alias for a load module but not the load module itself (ALIAS is specified on the ++DELETE statement), you must reflect this change using JCLIN. To do this, include a ++JCLIN statement with JCLIN data that contains a link-edit step for the load module, with the alias deleted from the list of aliases on the link-edit ALIAS statement. This causes SMP/E to replace the alias list in the CSI.

Packaging Recommendations

A base function should not contain a ++DELETE statement, unless a PTF containing the statement was integrated into a service update of that function.

New releases of a base function do not own elements that would need to be deleted from a library. However, there may be shared load modules that should be deleted. In this instance, a base function may contain a ++DELETE for the shared load module.

Although a program object residing in a PDSE can have an alias name greater than 8 characters, the ++DELETE statement cannot be used to delete such an alias value without deleting the program object. Instead, you need to resupply JCLIN to define the program object without providing an ALIAS statement for the alias value to be deleted. Make sure to also include a ++MOD statement for a module in the load module to force SMP/E to relink the load module.

End of Packaging Recommendations

If a load module resides in two or more system libraries, you need only one ++DELETE statement. Refer to the *SMP/E Reference* manual for information about the ++DELETE statement.

8.4 Deleting Elements from Libraries and SMP/E Data Sets

The DELETE operand on an element MCS indicates that the element is to be removed from the target libraries, distribution libraries, and SMP/E data sets. This operand can be used for all element types. This is an optional operand and is used only in dependent functions.

Packaging Rules (DELETE for Elements)

- □ Rule 123. A dependent function must not delete a macro or source element from a lower-level function (its parent base function or a lower-level dependent function for the same parent base function), because a PTF that is applicable to the lower-level function may need to update the element (such as by using ++MACUPD or ++SRCUPD). If that element were deleted, there would be nothing to update, and the PTF needed for the lower-level function could not be installed.

8.5 Enabling Load Module Changes at the CSECT Level (++MOD CSECT)

The CSECT operand lists all the CSECTs that are contained in a module. Defining the contents of a load module by CSECT name allows SMP/E to change a load module at the CSECT level when a function or module is being deleted.

Packaging Rules (++MOD CSECT)

- □ Rule 124. If a SYSMOD changes the CSECTs contained in an existing module, CSECT must be specified and must list all the CSECTs in that module. This is true even if the module now contains only one CSECT whose name matches the module name on the ++MOD statement.
- □ Rule 125. This rule has been deleted. It has been replaced by rule 142.5 in 9.6, "Link-Edit Steps" on page 87.

Packaging Recommendations

If CSECT is specified, it must include all the CSECTs contained in the module, even if one of them has the same name as the module. If this is done, SMP/E can change the affected load module at the CSECT level when a function or module is being deleted.

End of Packaging Recommendations

Note: Simply ordering the INCLUDE statements is not sufficient to define the order of CSECTS, because SMP/E replaces CSECTs when relinking the load module and could change the order of the CSECTs.

Refer to 9.2, "When Do You Need JCLIN?" on page 82 for information about using JCLIN to define load modules.

8.6 Defining Ownership of Elements (++element VERSION)

The VERSION operand is required to establish which elements are functionally higher when different SYSMODs ship elements with identical element names and element types. For example, it could be used to add elements to a dependent function when those elements already belong to a lower-level dependent function. Or, it could be used when two language-support dependent functions contain a common element that was not translated (such as a CLIST).

Note: Although SMP/E uses the VERSION operand to determine the correct version of the elements to be installed, it does not use VERSION to determine the relationships of SYSMODs being installed. You must specify that information on the PRE or SUP operand of the ++VER statement.

This operand is optional for dependent functions. It is not allowed in base functions.

You may need to create a new version of an element that already exists in a product. For example, you may need to add a user function to or provide service for an existing element. There are two ways of providing a new version of an element:

1. **Dependent function.** A new dependent function, or a new release of an existing dependent function, can provide a new version of an element. Service is provided for the new version of the element throughout the currency of the new dependent function. Service for the previous version of the element must continue to be provided during the currency of the previous release of the dependent function.
2. **PTF.** A PTF can be used to create a new version of an element in a base function or a dependent function.

Packaging Rules (VERSION for Elements)

- □ Rule 125.1. VERSION is required to establish which elements are functionally higher when SYSMODs for different functions have elements with the same type and name in common. You must specify the lower-level function in the VERSION operand of the element statement in the SYSMOD associated with the higher-level function.
- □ Rule 126. The specified functions must be able to coexist with the specifying SYSMOD.
- □ Rule 127. The specified functions must contain the element described by the element statement.
- □ Rule 128. For dependent functions, VERSION must specify all the dependent functions that are functionally lower than the specifying function and include the element being versioned.

Packaging Rules (VERSION for Elements)

- ☐ Rule 129. If VERSION is also specified on a ++VER statement for this SYSMOD, the VERSION operand on the element statement overrides the VERSION values specified on the ++VER statement. However, the VERSION operand on the element statement is not additive; it does not automatically take over ownership from the functions specified on the ++VER VERSION operand. To take over ownership from any of the functions specified on the ++VER VERSION operand, you must repeat those values on the VERSION operand for the element statement.
- ☐ Rule 130. The VERSION operand must be specified on the element statement if individual elements may be versioned differently. The VERSION operand must be specified on the ++VER statement used if all elements affected by this SYSMOD are to be versioned the same way.

Packaging Recommendations

++*element* VERSION should be used only by different functions of the same product. If the VERSION operand is used by a function that is not part of the same product as the element it wants to assume ownership of, unpredictable results may occur. For example, if Product A owns an element and Product B uses VERSION to assume ownership of that element, it may not be clear which product should ship a given PTF for that element.

If use of the VERSION operand between two products is unavoidable, it is the responsibility of the development owner of Product B to ensure that the development owner of Product A understands and agrees to what has been done.

End of Packaging Recommendations

Chapter 9. Using JCLIN

JCLIN provides information to SMP/E about how to install a SYSMOD in the target and distribution libraries. JCLIN can be provided in several formats, such as assemble, copy, and link-edit steps. SMP/E processes these steps to determine the structure of the SYSMOD's elements. SMP/E builds and updates entries based on JCLIN data; however, it does not actually execute the JCLIN input.

To help you understand how to use JCLIN, this chapter describes:

- Providing JCLIN data for function SYSMODs
- When you need to use JCLIN
- General packaging rules for JCLIN data
- Assembler steps
- Copy steps
- Link-edit steps
- Examples of JCLIN data

For more information about JCLIN processing, see the *IBM SMP/E for z/OS Commands* manual.

9.1 Providing JCLIN Data for Function SYSMODs

There are several sources of JCLIN data:

- Data associated with a ++JCLIN statement
- Output from the SMP/E GENERATE command
- Stage 1 output JCL from a system, subsystem, or product generation

The output JCL from a generation procedure can be processed by the JCLIN command to update the CSI target zone with information about the products installed by that JCL. However, once the JCLIN command has processed that JCLIN data, the product information cannot be removed from the target zone unless the product is deleted or restored.

To avoid any potential problems this might cause your customers, package JCLIN data using a ++JCLIN statement. When customers apply a SYSMOD containing a ++JCLIN statement, SMP/E saves unchanged copies of target zone entries that will be updated by the JCLIN. This way, if customers need to restore the SYSMOD, they can do it because SMP/E saved the previous version of the entries.

For more information about JCLIN processing, see the *IBM SMP/E for z/OS Commands* manual.

Notes:

1. JCLIN data is processed only for macros, modules, and source. It is not processed for data elements, except to define totally copied libraries. It is not processed, and should not be specified, for elements installed in a *hierarchical file system (HFS) or Java Archive (JAR) file*. Such elements are defined by the ++HFS or ++JAR statement.
2. SMP/E has no column limitations for operands beyond the normal JCL rules.

© Copyright IBM Corp. 1986, 2003

3. The ++JCLIN statement does not cause SMP/E to update the target or distribution libraries; only the entries in the target and distribution zones are updated. These libraries are updated when SMP/E processes the elements in the SYSMOD. The element statements in the SYSMOD determine which elements should be installed.

9.2 When Do You Need JCLIN?

You need JCLIN for a base function so that SMP/E has information about the structure of the product and target libraries:

- The library in which an element resides
- How modules are link-edited together for load modules
- Where the load modules exist and their characteristics

You also need JCLIN for changes introduced by a dependent function. You do not need to use JCLIN for structures and attributes that were not altered by the dependent function. This means you do not need JCLIN for every element; JCLIN is required only for those load modules with changed structure or attributes. Repeating JCLIN for unchanged elements increases the risk of errors.

Following are some situations that do not require JCLIN to be used:

- All elements (other than ++MOD elements that are totally copied modules) of a product are installed using a copy utility.
- A dependent function does not introduce new elements.
- A dependent function does not change the link-edit attributes for a load module.

Note: You never need JCLIN for data elements. SMP/E uses the SYSLIB and DISTLIB operands on the data element MCS to determine where the elements should be installed. The same is true for HFS and JAR elements. SMP/E uses the SYSLIB and DISTLIB operands on the ++HFS or ++JAR statements to determine where the elements should be installed.

You need JCLIN when a dependent function does any of the following:

- Changes the link-edit attributes of a load module

  The attributes of a load module include such things as RENT, REUS, and REFR.

- Changes the structure of a load module

  Structure means the ENTRY, ORDER, and ALIAS statements that apply to a load module.

- Introduces a new load module

Refer to Table 10 on page 71 for information about using MCS to perform operations on elements and load modules. For example, you do not need JCLIN when a dependent function or PTF is adding a module to an existing load module. Instead, you should use the LMOD operand on the ++MOD statement.

9.3 General Packaging Rules for JCLIN Data

The following general rules define how JCLIN data must be coded.

Packaging Rules (JCLIN Data)

- 131. The combination of JCLIN data and element statements must completely describe all the elements in the function and their target and distribution libraries.

□ Rule 131.1. A product's installation must not require the editing of the JCLIN.

□ Rule 131.2. JCLIN data must be provided in a RELFILE for function SYSMODs, instead of inline after the ++JCLIN statement.

□ Rule 132. If the low-level qualifier of a data set name is in the format *xccczzzz*, as described in rule 149, the low-level qualifier and the ddname must be identical.

Exception: Since a DDNAME may refer to a subdirectory in the HFS or JAR, several DDNAMEs may point into one HFS or JAR. In these cases, the low-level qualifier and the DDNAME need not be identical. (This is the same as rule 150 in 10.4, "Library Names" on page 117.)

□ Rule 133. Input data sets in link-edit steps must not be concatenated, with the exception of the //SYSLIB DD statement used with CALLIBS support. For more information, see the description of the SYSLIB DD statement in 9.6.2, "Link-Edit Control Statements" on page 90.

In addition to the above rules, the following recommendations apply to JCLIN data:

Packaging Recommendations

- Use the simplest possible JCL statements.
- Specify all information in uppercase (verbs as well as values).

  This is necessary to avoid syntax errors or incorrect results during SMP/E processing.

  Note: This convention does not apply to values on the ALIAS statement. These values must be specified in the desired case (uppercase or mixed-case), because they are used as is.
- Do not use update steps in JCLIN data; SMP/E ignores them.
- In the JCLIN of the dependent function, describe only new or changed structure.

  The JCLIN for a dependent function should not repeat data already provided in the JCLIN of the base function.
- Do not use abbreviations.

  SMP/E may not recognize all abbreviations.
- For copied members (except for ++MOD), use the SYSLIB and DISTLIB operands on the element statements instead of JCLIN to define copies.

- If possible, do not use continued utility control statements. Although SMP/E tries to support all existing formats of the utility control statements, it cannot completely duplicate the syntax checking of the utility. The safe method is to use the simplest format of the utility control statement without continuations.
- Test the JCLIN data as follows:
  - Perform RECEIVE, APPLY, and ACCEPT of the product on one system.
  - Perform RECEIVE, ACCEPT BYPASS(APPLYCHECK), GENERATE of the product on a second system.
  - Compare the SMP/E reports from the two products, checking for discrepancies.
  - Compare every library, member by member, between the two products, checking for discrepancies.
  - Run the JCLIN data outside of SMP/E and compare the resulting load modules with those built during the SMP/E installs. There should be no differences.

End of Packaging Recommendations

9.4 Assembler Steps

Packaging Rules (JCLIN Assembler Steps)

- ☐ Rule 134. Assembler steps must be identified by one of the following:
  - EXEC PGM=IFOX00
  - EXEC PGM=IEV90
  - EXEC PGM=ASMA90
  - EXEC PGM=ASMBLR
  - EXEC ASMS

9.5 Copy Steps

Packaging Rules (JCLIN Copy Steps)

- ☐ Rule 135. Copy steps must be identified by the following:
  - EXEC PGM=IEBCOPY
- ☐ Rule 136. The RENAME function must not be used in JCLIN.

Packaging Rules (JCLIN Copy Steps)

- ☐ Rule 137. If the SELECT MEMBER= statement is used to selectively copy elements, the COPY INDD=*xxx*,OUTDD=*xxx* control statements for selectively copied elements must include the comment TYPE=*xxxx*. The format of the TYPE comment on the COPY statement is:

  ```
  COPY INDD=ddname,OUTDD=ddname   TYPE=xxxx
  ```

  where *xxxx* is MOD, MAC, SRC, or DATA.

  Notes:

  1. If the TYPE=*xxxx* parameter is not specified, the default used by SMP/E is TYPE=MOD.
  2. TYPE=DATA is used for data elements.

  Without this additional comment, the GENERATE command cannot determine what type of element is being copied. If the comment is not included, SMP/E assumes the element is a module and may create unnecessary module entries in the target or distribution zone.

  For data elements, HFS and JAR elements, you must use the SYSLIB and DISTLIB operands on the element statement to specify information used to install the element. During JCLIN processing, SMP/E bypasses any COPY SELECT statements that specify TYPE=DATA.

- ☐ Rule 138. The SELECT statement can specify either the name of the member to be copied or an alias name for the member. The SELECT statement for an alias must specify the comment "ALIAS OF *member*", where *member* is the member name for which *alias* is an alias.

- ☐ Rule 139. A SELECT statement that identifies an alias can specify only one name on the MEMBER operand.

- ☐ Rule 139.1. If a ++MOD on a product tape defines either (1) a complete load module containing single or multiple CSECTs or (2) a partial load module containing multiple CSECTs, any ++MOD by the same name shipped in a subsequent PTF must also be the same type of load module (complete load module or multi-CSECT partial load module). If a CSECT shipped in the original ++MOD is not shipped in the replacement ++MOD, it will no longer exist.

  To replace part of a copied ++MOD, the PTF must convert the ++MOD into a link-edited load module by splitting it into smaller serviceable parts, as follows:

  1. Delete the original ++MOD with a ++MOD DELETE.
  2. Ship a new ++MOD for each of the parts into which the original ++MOD has been split.
  3. Provide link-edit JCLIN to define the link edit structure of the resulting load modules.

  All future maintenance that affects the load module or any of its parts must explicitly or implicitly specify this PTF as a prerequisite.

Packaging Recommendations

Although JCLIN can be used to identify copied elements, the preferred way of copying elements other than ++MODs is to specify the DISTLIB and SYSLIB operands on the element MCS.

End of Packaging Recommendations

Notes:

1. Copy input must be **inline**, not pointing to another data set.
2. The only copy utility control statements allowed are COPY (or C) and SELECT (or S).

9.5.1 Considerations for the SELECT Statement for Copy Operations

When deciding whether to specify a SELECT statement in your copy steps, you need to consider how SMP/E processes copy steps:

- A COPY without SELECT MEMBER creates SMP/E DLIB entries.
- A COPY with SELECT MEMBER does not create the DLIB entries, but it either updates the SYSLIB subentry for MAC and SRC entries or builds MOD or LMOD entries (with the COPY indicator turned on) for modules.

Following are recommendations for IEBCOPY steps in product JCLIN:

- Do not use SELECT MEMBER statements for elements that are fully defined in the SMPMCS.

  Data elements, HFA and JAR elements must be fully defined in the SMPMCS.
- Use COPY statements with SELECT MEMBER statements for single-CSECT load modules that can be copied.
- Use COPY statements (without SELECT MEMBER statements) for each totally copied library.

9.5.1.1 Fully-Defined Elements

Copy steps are not required for fully-defined elements. Instead, the element statement should specify both DISTLIB and SYSLIB for all elements except ++MOD. Here are some examples:

```
++MAC(xxxxxxxx) DISTLIB(AMACLIB) SYSLIB(MACLIB) .
++SRC(xxxxxxxx) DISTLIB(AJES3SRC) SYSLIB(JES3SRC) .
++PROC(xxxxxxxx) DISTLIB(APROCLIB) SYSLIB(PROCLIB) .
++HFS(xxxxxxxx) DISTLIB(ABPXLIB) SYSLIB(BPXLIB) .
++JAR(xxxxxxxx) DISTLIB(ABPXLIB) SYSLIB(BPXLIB) .
```

9.5.1.2 Single-CSECT Load Modules

Copy steps should be used for single-CSECT load modules that can be copied. Here is an example:

```
COPY INDD=ALINKLIB,OUTDD=LINKLIB  TYPE=MOD
  SELECT MEMBER=xxxxxxxx
  SELECT MEMBER=yyyyyyyy
```

This statement indicates that SMP/E should build an LMOD entry with the same name as the module. In this LMOD entry, the COPY indicator should be turned on and the SYSLIB subentry should be LINKLIB.

9.5.1.3 Totally Copied Libraries

When no SELECT statement is specified for a copy step, SMP/E creates DLIB entries, which it uses to determine the appropriate target library (if none was specified on the element MCS and one didn't already exist). The DLIB entry indicates that the library specified in the INDD= parameter is totally copied to the library specified in the OUTDD= parameter. Here is an example:

```
COPY INDD=AMACLIB,OUTDD=MACLIB    TYPE=MAC
```

This statement indicates to SMP/E that if an element being processed has a distribution library of AMACLIB but does not specify a SYSLIB of MACLIB--for example, ++MAC(xxxxxxxx) DISTLIB(AMACLIB) -- SMP/E should install the macro in MACLIB and add the SYSLIB subentry of MACLIB to the macro entry.

Similar processing occurs for modules--except that the SYSLIB subentries are in LMOD entries (not MOD entries). Here is an example:

```
COPY INDD=ALINKLIB,OUTDD=LINKLIB
```

This statement indicates to SMP/E that if a module being processed has a distribution library of ALINKLIB but is not yet associated with a load module, SMP/E should build an LMOD entry with the same name as the module. In this LMOD entry the COPY indicator should be turned on and the SYSLIB subentry should be LINKLIB.

Packaging Recommendations

If you develop a new release of a function that uses totally copied libraries, and the new release copies the distribution library into a different target library from the previous release, you should instruct the users to delete the DLIB entry from the CSI before they apply or accept the new release. This ensures that when SMP/E installs the new release, it builds new DLIB entries pointing to only the new target library.

End of Packaging Recommendations

9.6 Link-Edit Steps

Packaging Rules (JCLIN Link-Edit Steps)

- □ Rule 140. Link-edit steps must be identified by one of the following:
  - EXEC PGM=IEWL
  - EXEC PGM=HEWL
  - EXEC PGM=IEWBLINK
  - EXEC LINKS

Packaging Rules (JCLIN Link-Edit Steps)

- ☐ Rule 141. Link-edit steps must not be sensitive to the order of execution of other link-edit steps, either in the same FMID or in another FMID. Link-edit steps must also not be sensitive to the order of execution of the individual load module builds within the step.
- ☐ Rule 142. No elements to be included in a JCLIN link-edit step can be derived from the output of another JCLIN link-edit step, or from the output of a load module build within the same JCLIN link-edit step.
- ☐ Rule 142.1. Never specify a JCLIN link-edit step to indicate that a load module resides in the SMPLTS library.

  SMP/E automatically link-edits a base version of any load module with a CALLLIBS subentry into the SMPLTS library.
- ☐ Rule 142.2. Do not specify a pathname in a hierarchical file system (HFS) or Java Archive (JAR) file as the distribution library.
- ☐ Rule 142.3. All INCLUDE statements for modules in link-edit JCLIN data must specify the included module's distribution library, or SYSPUNCH if it is an assembled module. Do not use data sets such as SYSLIB or SYSLMOD.

  All INCLUDE statements for utility input (using the TYPE=UTIN comment) such as side decks must specify the utility input file's target library.
- ☐ Rule 142.4. If a load module consists of more than one distribution library module, use an ENTRY statement; otherwise, the entry point of the load module might change each time the load module is relinked by SMP/E.
- ☐ Rule 142.5. If a specific order of CSECTs within a load module is required, use ORDER statements to define the load module structure.
- ☐ Rule 142.6. If Product A uses CALLLIBS to indicate libraries created by Product B:

  1. The SYSLIB DD statement in Product A's JCLIN must use the real DDNAME of the library.
  2. Product A's DDDEF job must, in a seperate step, provide ADD DDDEF commands to create DDDEF entries for all CALLLIBS libraries with their real DDNAMEs. The job instructions must explain that these ADD DDDEF commands can be commented out or removed it the entries already exist in the zone.
  3. If it is possible that the library may not exist on the system, the DDDEF job must instruct the customer to point either to the actual dataset (if it exists) or to an empty dataset. The job must not give the customer a choice of two or more legitimate datasets for one DDDEF. NOTE: The product may not allocate an empty dataset for this purpose.
- ☐ Rule 142.7. If a product documents in its Program Directory that a Return Code 8 is acceptable from APPLY, then RC=8 must be coded on the NAME statement in the JCLIN for the appropriate load module(s). This may be the case if the product uses a CALLLIBS library to obtain load modules created by an optional function.

Packaging Rules (JCLIN Link-Edit Steps)

☐ Rule 142.8. For all modules appearing in all INCLUDE statements in the JCLIN for SYSMOD A, one of the following must be true:

1. The module is shipped as a ++MOD element in SYSMOD A.
2. The module is shipped as a ++MOD element in a SYSMOD identified in the FMID, REQ, or PRE operands on SYSMOD A's ++VER MCS.

NOTE: This rule does not apply to INCLUDE statements for utility input files (using the TYPE=UTIN comment).

SMP/E does not order the link-edit steps based on the order specified in the JCLIN. Instead, if multiple load modules and target libraries are involved, SMP/E organizes the link-edit steps for the most efficient invocations of the link-edit utility (which might not be the same order as the JCLIN data). For example, assume that a product consists of a base function and a dependent function.

- The dependent function conditionally coexists with the base function; it can be installed with the base function but is not a prerequisite for the base function.
- The base function must have its own JCLIN data that completely describes the elements it contains, because a user may choose to install the functions together or separately.

  If the base function is separately installed, its JCLIN data cannot contain a link-edit step including elements from the dependent function, because those elements are not yet available.

Packaging Recommendations

- Product A should not INCLUDE modules created by Product B unless all of the following are true:
  1. Product B is guaranteed to always be present in the same target zone as Product A
  2. The module always exists in the same library, no matter which release of any product is present
  3. The library containing the module is always guaranteed to exist

  If any of these conditions are not true, the product should use CALLLIBS instead of explicit INCLUDEs.
- The LMOD RC parameter should be specified on every JCLIN NAME statement. The value for each load module should match the expected return code from link-editing that load module, and the highest value within the JCLIN for an FMID should match the expected APPLY return code documented in the Program Directory.

End of Packaging Recommendations

9.6.1 JCLIN Processing of DD Statements in Link-Edit Steps

Target libraries should be identified in link-edit steps by the SYSLMOD DD statement. All other DD statements for target libraries are ignored. JCLIN processing extracts the lowest-level qualifier from the data set name on the SYSLMOD DD statement, uses that qualifier as a ddname, and passes the link-edit utility a DD statement allocated to the data set with that ddname. For example, when JCLIN processing encounters this DD statement:

```
//SYSLMOD DD DSN=PROD1.V1R1M0.SABCMOD1
```

it searches the target zone for a DDDEF entry with the name SABCMOD1. The data set name identified in that DDDEF entry is passed to the link-edit utility as the output (SYSLMOD) data set.

DD statements for distribution libraries are ignored by JCLIN processing. The ddnames specified on INCLUDE statements in the JCLIN are used as the DISTLIB value in the MOD entries that are created. For example, when JCLIN processing encounters this statement:

```
INCLUDE AABCLOAD(ABCMOD01)
```

it builds a MOD entry for ABCMOD01 and indicates a DISTLIB value of AABCLOAD. For more information about link-edit control statements, see 9.6.2, “Link-Edit Control Statements.” For details about JCLIN processing, see the *IBM SMP/E for z/OS Commands* manual.

9.6.2 Link-Edit Control Statements

All required link-edit control statements must be specified. This section describes considerations for specific link-edit statements. Here are some special considerations to keep in mind:

- If a load module consists of more than one distribution library module, use an ENTRY statement; otherwise, the entry point of the load module might change each time the load module is relinked by SMP/E.
- If a specific order of CSECTs within a load module is required, use ORDER statements to define the load module structure. See 8.5, “Enabling Load Module Changes at the CSECT Level (++MOD CSECT)” on page 77 for more information.
- If PLISTART is listed first in a PL/I load module, ORDER cards do not work; any requirement for ORDER should be changed to ENTRY.

ALIAS statement

To ensure that SMP/E can process your link-edit ALIAS control statements, you must address the following considerations:

- **General considerations**
  - An ALIAS control statement can span any number of 80-byte records.

    Note: If you assign a load module residing in a PDSE an alias value greater than eight characters, you cannot later use the ++DELETE statement to delete that alias value (and not the associated load module). To delete such an alias value without deleting the load module, you need to resupply JCLIN to define the load module without providing an ALIAS statement for the

alias value to be deleted. Make sure to also include a ++MOD statement for a module in the load module to force SMP/E to relink the load module.

– Column 1 of all 80-byte records composing an ALIAS control statement must contain a blank (X'40').
– The data for the first 80-byte record of an ALIAS control statement must start in column 2 or later and end anywhere up to and including column 71.
– The control statement type (ALIAS) must be followed by at least 1 blank (X'40').
– The control statement type (ALIAS) must be in uppercase.
– Columns 73 through 80 of an 80-byte record are ignored.
– An alias value can be from one to 64 characters.
– An alias value can be composed of characters in the range X'41' through X'FE'.

Note: Although the binder also accepts characters X'0E' (shift-out character) and X'0F' (shift-in character), SMP/E does not accept them.

– An alias value can be enclosed in single apostrophes. It must be enclosed in single apostrophes in the following cases:
  - It contains a character other than uppercase alphabetic, numeric, national ($, #, or @), slash, plus, hyphen, period, and ampersand. Here is an example:

```
        1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS 'This_alias_contains_special_characters!!!!'
```

  - It is continued to another 80-byte record of the control statement. Here is an example:

```
        1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS        'This_is_a_very_long_value_that_is_continued_to_the_next*
 _card!'
```

– If an apostrophe is part of the alias value (not a delimiter), two apostrophes need to be specified in the appropriate location in the alias value. These two apostrophes count as two characters in the 64-character limit for an alias value. Here is an example:

```
        1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS 'It''s_the_quote_that_makes_apostrophes_necessary.'
```

– The single apostrophes used to enclose an alias value do not count as part of the 64-character limit for an alias value. For example, the alias value in the following example contains 10 characters:

```
        1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS  'Only_ten!!'
```

– SMP/E uses the alias value exactly as is. SMP/E does not try to enforce any rules the binder may be using as a result of the CASE execution parameter.

Warning to Packagers

Be extremely careful when creating the JCL and link-edit ALIAS control statements to be processed by SMP/E as JCLIN. When parsing the ALIAS control statements to derive alias values, SMP/E does not try to replicate binder processing that would result from a particular specification of the CASE execution parameter. Therefore, you must ensure that the values on the ALIAS control statement are exactly as desired and that the proper CASE value is used so that the link-edit utility produces the desired results.

End of Warning to Packagers

- **Continuation records**
  - Column 72 of a given 80-byte record must be a nonblank character if the control statement is continued onto the next 80-byte record. The character in column 72 denotes only continuation and is never part of an alias value.
  - The data for continuation records (80-byte records 2 through *n* of an ALIAS control statement) can start in column 2 or later and end anywhere up to and including column 71 (for example, if multiple aliases are being specified).

    The data for a continuation record must start in column 2 if it is part of an alias value that is being continued from the previous 80-byte record. An alias value that is continued from one 80-byte record to another 80-byte record must do all of the following:

    - Be enclosed in single apostrophes
    - Extend through column 71 of the first 80-byte record
    - Start in column 2 of the next 80-byte record
    - Have a nonblank continuation character in column 72

    Here is an example:

```
        1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS         'This_is_a_very_long_value_that_is_continued_to_the_next*
 _card!'
```

- **Entry points**

  A new format of the ALIAS statement supported for the binder allows an alternative entry point to be specified into a load module. If this new format is used, each alias name with an associated entry point must be specified on its own 80-byte record, with a separate ALIAS statement; no other aliases should be specified on that statement. If multiple alias values of this format are specified on a single ALIAS control statement, only the first is recognized; the rest are ignored.

  Note: When this form of the ALIAS control statement is used, the alias value cannot be 64 characters long, because SMP/E requires the statement to be complete on one 80-byte record. When this form of the ALIAS control statement is used, the maximum length for an alias value is 61 characters.

Suppose that a load module has the following aliases: ALA1, ALA2, ALA3, and ALA4. ALA1 and ALA2 are associated with entry point names ENTRYPT1 and ENTRYPT2, respectively.

– Here are examples of how the aliases should be specified:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALA1(ENTRYPT1)
 ALIAS ALA2(ENTRYPT2)
 ALIAS ALA3
 ALIAS ALA4
or
 ALIAS ALA3,ALA4
```

– Here are examples of how the aliases should *not* be specified:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALA1(ENTRYPT1),ALA2(ENTRYPT2),ALA3,ALA4
or
 ALIAS ALA1(ENTRYPT1),ALA3
 ALIAS ALA2(ENTRYPT2),ALA4
```

- **Multiple aliases**

  – Multiple alias values can be specified on a single ALIAS control statement as long as they are not in the form alias(entrypoint). Multiple alias values must be separated by commas (“,”). Here is an example:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALIAS1,ALIAS2,ALIAS3,ALIAS4
```

  – Multiple alias values can span multiple 80-byte records. When this occurs, there must be a nonblank character in column 72, and one of the following must be true:

    - The last alias value on the 80-byte record that is being continued must be followed by a comma and one or more blanks (“, ...”). Here is an example:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALIAS1,ALIAS2,                                                   *
   ALIAS3,ALIAS4
```

    - The last alias value on the 80-byte record that is being continued must be followed by a comma (“,”) in column 71. Here is an example:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALIAS1,ALIAS2,'A_relatively_long_ALIAS_but_not_quite_64_chars.',*
   ALIAS4,ALIAS5
```

    - The last alias value on the 80-byte record that is being continued can be enclosed in single apostrophes such that part of the alias value appears on the current 80-byte record and part appears on the next 80-byte record (see the rules for continuation records). Here is an example:

```
         1         2         3         4         5         6         7         8
----+----0----+----0----+----0----+----0----+----0----+----0----+----0----+----0
 ALIAS ALIAS1,ALIAS2,'A_relatively_long_ALIAS.',ALIAS4,'Not_too_long_bu*
 t_wraps.',ALIAS5,ALIAS6
```

  – If a blank (X'40') follows an alias value, SMP/E assumes there are no more alias values for the current ALIAS control statement.

ENTRY statement
Each load module consisting of more than one distribution library module must have an ENTRY statement; otherwise, the entry point of the load module changes each time the load module is relinked by SMP/E.

EXPAND statement

Packaging Recommendations

EXPAND statements should not be used in JCLIN data, because they would be saved in the LMOD entry and would cause the load module to be expanded each time it is link-edited. This is not always desirable.

End of Packaging Recommendations

IDENTIFY statement

Packaging Recommendations

IDENTIFY statements should not be used in JCLIN data. They are produced as part of servicing a module. If found in the JCLIN, they are stored in the LMOD entry and can result in incorrect data being stored during the application of service.

End of Packaging Recommendations

INCLUDE *ddname***(***member,member***...) statement**
INCLUDE statements are used to identify the modules in the load module. They are also used to identify utility input to be included when the load module is link-edited. This is denoted by the TYPE comment on the INCLUDE statement. The format of the TYPE comment on the INCLUDE statement is:

INCLUDE *ddname***(***member,member***...) TYPE=UTIN**

There must be at least one blank between the closing parenthesis of the INCLUDE statement and the TYPE comment. If the TYPE comment is not specified, SMP/E assumes the INCLUDE statement identifies modules.

Processing Modules:

Each module in the load module must be specified as a member name on an INCLUDE statement.

The member names are assumed to be modules existing in distribution library *ddname*. SMP/E builds MOD entries for each member name specified and sets the DISTLIB value in each MOD entry to *ddname*. (An exception to this is when the ddname is SYSLMOD. In that case, no MOD entry is built for the INCLUDE statement.) SMP/E does not refer to the *ddname* DD statement to determine the actual library referred to. Therefore, all ddnames specified on INCLUDE statements must be the actual ddnames assigned to the products.

The INCLUDE statements are not saved in the LMOD entry, because they are not necessary when the load module is link-edited. All link-edits requested by SMP/E are CSECT-replaces; the load module is built from the new version of the updated CSECT and the existing load module from the target library.

The ddnames *SYSPUNCH* and *SMPOBJ* are reserved for inclusions of object decks produced by assembly steps that are not to be link-edited to a distribution library during ACCEPT processing. In both cases, the name stored in the MOD entry's DISTLIB subentry is SYSPUNCH.

Processing Utility Input:

Each utility input file must be specified as a member on an INCLUDE statement with the TYPE=UTIN comment.

The member names are assumed to be members of the library *ddname*. SMP/E builds a utility input subentry for each member name specified and stores it into the LMOD entry. Each utility input subentry contains the member name and the ddname. The utility input files will be included when link-editing the load module. These files may identify definition side decks containing link-edit control statements, or any other file to be included during a link-edit operation.

Note: For a product including modules not provided by that product, the INCLUDE statements must specify either a distribution library ddname or SYSPUNCH. (SYSPUNCH is used only for processing assembled modules.) If your product will be installed in the same target and distribution zones as the other product, see 9.6.4, “Cross-Product Load Modules for Products Installed in the Same Zone” on page 100 for more information. If you cannot be sure, or if you know that your product will be installed in different target and distribution zones from the other product, see 9.6.5, “Cross-Product Load Modules for Products Installed in Different Zones” on page 102 for more information.

INSERT and OVERLAY statements

If a load module is to be linked in overlay structure, you must supply an INSERT control statement for each CSECT in the load module, including INSERT statements for those CSECTs within the root segment. It is not sufficient to properly place the INCLUDE and OVERLAY control statements.

LIBRARY statement

Normally, LIBRARY statements should not be used in JCLIN data. An exception is when the CALLLIBS operand is specified on the JCLIN command or ++JCLIN MCS, or when //*CALLLIBS=YES is encountered after a job card preceding a link-edit step. JCLIN processing then allows for the LIBRARY state-

ment to be used to specify those modules (external references) that are to be excluded from the automatic library search during the following:

- The current linkage editor job step (restricted no-call function)
- Any subsequent linkage editor job step (never-call function)

A LIBRARY statement should be used only if a SYSLIB DD statement is also used. It should not be used to specify additional automatic call libraries; the SYSLIB DD statement should be used instead.

NAME *lmodname*(R) statement
When SMP/E encounters either the NAME control statement or the end of input with no NAME statement, SMP/E builds an LMOD entry. How SMP/E determines the name of the LMOD depends on the JCL being scanned:

- If the NAME statement is found, SMP/E gets the LMOD name from the *lmodname* field of the NAME statement.
- If no NAME statement is found and a SYSLMOD DD statement is present, SMP/E gets the LMOD name from the member name of the data set specified. If no member name is specified, SMP/E issues an error message identifying the JOBNAME and STEPNAME and the reason for the error.
- If no NAME and SYSLMOD DD statements are found, SMP/E searches for the MOD=*name* operand in the JCL and uses that name as the LMOD name. If no MOD=*name* operand is found, SMP/E issues an error message.

ORDER statement
If a specific order of CSECTs within a load module is necessary, ORDER statements are required to define the load module structure. Simply ordering the INCLUDE statements is not sufficient, because SMP/E does CSECT replacements when relinking the load module and, therefore, changes the order of the CSECTs.

REPLACE statement
REPLACE statements are saved in the LMOD entry and are associated with the DLIB module name found on the next INCLUDE statement in the JCL. If the same INCLUDE statement is processed later by JCLIN, REPLACE statements already in the LMOD entry associated with this INCLUDE statement are deleted and replaced by any associated REPLACE statements in the latest job. REPLACE statements are passed to the linkage editor only when the associated DLIB module is to be replaced in the load module.

SYSDEFSD DD statement
SMP/E uses the SYSDEFSD DD statement to determine the side deck library for the load module. SMP/E determines the ddname by using the lowest-level qualifier of the data set name specified in the SYSDEFSD DD statement. This ddname is saved as the side deck library subentry in the LMOD entry.

The side deck library will contain the definition side deck for the load module created by the link-edit utility. The definition side deck contains link-edit control statements describing the load module.

SYSLIB DD statement
Normally, SYSLIB DD statements should not be used in JCLIN data. However, they can be used for load modules needing to implicitly include modules from other products. Such load modules are commonly used by products that:

- Are written in a high-level language and, as a result, include modules from libraries (such as compiler libraries) that are owned by a different product
- Make use of a callable-services interface provided by another product
- Need to include stub routines or interface modules from different products that may reside in other zones

For such load modules, the SYSLIB DD statement should specify all the automatic call libraries SMP/E is to use when linking the load module. (These libraries should be target libraries.) The low-level qualifier of each data set specified in the SYSLIB concatenation is saved as a CALLLIBS subentry for the associated load module. For SMP/E to link implicitly-included modules from these libraries, the user must provide DDDEF entries for the libraries in the zone containing the LMOD entry.

SYSLIB DD statements are processed only if the CALLLIBS operand is specified on the JCLIN command or ++JCLIN MCS, or if //*CALLLIBS=YES is encountered after a job card preceding a link-edit step. If the CALLLIBS operand or the CALLLIBS comment is not specified, SMP/E ignores any SYSLIB DD statements it encounters.

Implementation Notes:

- It is best to use this SYSLIB support when introducing a new version or release of your product, or when introducing a new load module for an existing version or release of your product.

  Using this SYSLIB support for an existing load module in a current product is not recommended. However, if you need to make such a change, make sure to do the following in the SYSMOD introducing the change:

  1. Supply a JCLIN link-edit step to redefine the load module. This step must specify the SYSLIB allocation needed for the load module.
  2. Specify the CALLLIBS operand on the ++JCLIN statement to ensure that the SYSLIB DD statement is processed.
  3. Supply all the modules that are explicitly included in the JCLIN link-edit step and that are owned either by this SYSMOD or by its FMID.

     Modules that are explicitly included in the JCLIN link-edit step and that are not owned by this SYSMOD or its FMID are included by SMP/E through normal load module build processing.

  If the existing load module had included cross-zone modules through the use of the LINK command, those modules are no longer included in the load module after the installation of the SYSMOD that redefined the load module. In this case, SMP/E issues warning messages. After the installation of the SYSMOD, the user must rerun the LINK command to include those cross-zone modules back into the load module. For more information about the LINK command, see the *SMP/E Commands* manual or the *SMP/E Release 7 Reference* manual.

- When a load module is built using SYSLIB DD statements, SMP/E cannot completely service the load module because it does not know the content of the load module. Specifically, the load module is not automatically rebuilt when an implicitly-included module is serviced. However, users can run the REPORT CALLLIBS command to identify and relink such load

modules. For more information about the REPORT CALLLIBS command, the *SMP/E Commands* manual or the *SMP/E Release 7 Reference* manual.

Including Pathnames in a SYSLIB Concatenation: A DD statement in a SYSLIB concatenation can include the PATH operand to specify a pathname as an automatic call library. A LIBRARYDD comment statement must immediately follow this DD statement and specify the ddname to be associated with that pathname. SMP/E saves the ddname specified on the LIBRARYDD comment as part of the CALLLIBS list in the LMOD entry being updated or created. For an example, see 9.7.5, "JCLIN Data for Load Modules Residing in a Hierarchical File System or Java Archive file" on page 112.

Notes:

1. If a DD statement in the concatenation comes between the DD statement specifying the PATH operand and the LIBRARYDD comment, the misplaced DD statement is ignored.
2. If the DD statement specifying the PATH operand is followed by a JCL statement other than a LIBRARYDD comment or a continuation DD statement for the SYSLIB concatenation, the LMOD entry is not updated or created. In addition, if the JCLIN was specified in a SYSMOD (instead of being processed by the JCLIN command), processing for that SYSMOD fails.

SYSLMOD DD statement

SMP/E uses either the SYSLMOD DD statement or the NAME statement to determine the target library for the load module, as follows:

- If a SYSLMOD DD statement is present, SMP/E determines the target library ddname by using the lowest-level qualifier of the data set name specified in the SYSLMOD DD statement.
- If no SYSLMOD DD statement is present, SMP/E determines the name by looking at the NAME=*dsname* option on the procedure statement. The ddname used is the lowest-level qualifier of the data set name specified in the NAME option.
- If no SYSLMOD DD statement or NAME=*dsname* value is found, SMP/E issues an error message.

The ddname of the target library is saved as the SYSLIB value in the LMOD entry for the load module.

A SYSLMOD DD statement can include the PATH operand to specify a pathname for installing a load module in a hierarchical file system. A LIBRARYDD comment statement must immediately follow this DD statement and specify the ddname to be associated with that pathname. SMP/E saves the ddname specified on the LIBRARYDD comment as a SYSLIB subentry in the LMOD entry being updated or created. For an example, see 9.7.5, "JCLIN Data for Load Modules Residing in a Hierarchical File System or Java Archive file" on page 112.

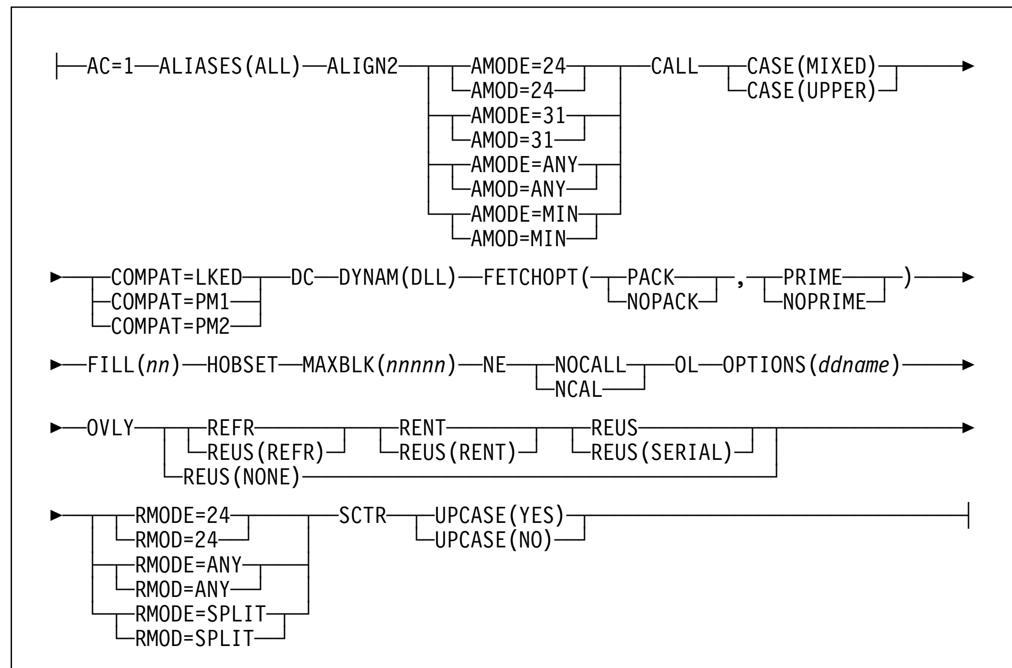
Notes:

1. If the DD statement specifying the PATH operand is followed by a JCL statement other than a LIBRARYDD comment, the LMOD entry is not updated or created. In addition, if the JCLIN was specified in a SYSMOD (instead of being processed by the JCLIN command), processing for that SYSMOD fails.
2. An LMOD entry can have at most two SYSLIB subentries. If the LMOD entry already contains two SYSLIB subentries, SMP/E replaces the second SYSLIB ddname with the ddname found on the SYSLMOD DD statement, the NAME=*dsname* option, or the LIBRARYDD comment statement.

All other statements found in link-edit input
All other link-edit control statements found are saved in the LMOD entry in the order they are encountered, and are passed to the linkage editor whenever SMP/E needs to relink this load module.

9.6.3 Link-Edit Attribute Parameters

These are the link-edit attributes SMP/E recognizes in the PARM field and saves for future processing:



When none of the above attributes are found, the STD indicator is set in the LMOD entry to indicate that the load module should be link-edited without any particular attributes.

Notes:

1. The OPTIONS attribute is recognized and processed, but it is not saved as part of the LMOD entry or the MOD entry being processed. It is used as a pointer to an imbedded file containing additional option specifications, allowing the PARM string to exceed the 100-character limit.
2. For more information on which attributes you can use with a specific link-edit utility, see the reference manual for that utility.

9.6.4 Cross-Product Load Modules for Products Installed in the Same Zone

There are two basic reasons for products to require cross-product load modules:

- A load module for one function SYSMOD needs to include a module from another function SYSMOD.
- A function SYSMOD needs to include some of its own modules in a load module of another function SYSMOD.

9.6.4.1 Linking a Module from Another Function

If a load module for Product A needs to include a module from Product B, a link-edit step in the JCLIN data for Product A must do one of the following:

- **Explicitly define the modules:** To explicitly define Product B modules to be included, the INCLUDE statement must be used.
- **Implicitly define the modules:** To implicitly define Product B modules to be included, the SYSLIB statement must be used. (The LIBRARY statement can also be used, if any specific modules should not be included.)

  For more information on this method, see 9.6.5.2, "Implicitly Defining the Modules" on page 103.

Table 11 briefly compares the two methods.

Table 11. Comparison of Explicit versus Implicit Definition of Modules

Consideration	Explicit Definition	Implicit Definition
Modules are automatically serviced.	X	
Modules do not need to be specified individually.		X

The function SYSMOD for Product A will not contain a ++MOD statement for the Product B module. If the Product B module is installed and the Product A load module does not already exist, the module is automatically included in the Product A load module.

If a load module for your product (Product A) requires a module from another product, you should describe this in the installation documentation for Product A and mention any additional jobs the user should run.

If the module from Product B is deleted (such as if a new replacement release of Product B is installed), SMP/E keeps a record of the fact that the module had been a part of a load module in Product A. As a result, if the module is reintroduced by

Product B (such as in the replacement release of the product), SMP/E automatically relinks the load module from Product A to include the module from Product B. If the module is not reintroduced but is still required in Product A, and a copy of the module is still available on the system, the user must use the JCLIN command to reprocess the JCLIN data for Product A and then rerun the postinstallation link-edit job.

On the other hand, a new release of Product A might delete the previous release of Product A and redefine the load module without including any of the borrowed modules. As part of installing the new release of Product A, SMP/E will first delete the old Product A modules from the load module, leaving a copy of the load module consisting solely of modules borrowed from other products. SMP/E will then use this copy of the load module (with the borrowed modules) as input when rebuilding the load module for the new release of Product A. To ensure that the new version of the load module does not include the borrowed modules, the new release of Product A must contain a ++DELETE MCS for the load module (to delete the previous version) in addition to the JCLIN needed to rebuild the new version of the load module.

9.6.4.2 Linking Modules into a Load Module for Another Function

If a function SYSMOD(1) needs to include any of its own modules in a load module of another function SYSMOD(2), you have two packaging options:

- If the load module already exists, and no link-edit control statements must be added or changed to add the modules to the load module, the ++MOD MCS for each module can specify the load module on the LMOD operand.
- If the load module does not exist, or if any link-edit control statements must be added or changed to add the module to the load module, the JCLIN data for function SYSMOD(1) must specify an INCLUDE statement for each of those modules followed by an INCLUDE SYSLMOD statement for the load module of function SYSMOD(2).

You can use these techniques to include a module for a dependent function in a load module for its parent base function, or to include a module for Product A in a load module for Product B. However, Product B must be installed before Product A.

A new release of Product A might no longer need to include its modules in a load module for Product B. However, because the new release of Product A deletes the previous release, SMP/E updates the LMOD entry for the Product B load module to track the modules that were deleted. As a result, if no action is taken, SMP/E relinks the Product A modules into the Product B load module when the new release of Product A is installed. You must make sure the installation documentation for your product tells the user how to avoid this problem. Here are the steps you need to describe:

1. Build a dummy function to delete Product A. (For an example, see the recommendations under 7.2.4, “Deleting SYSMODs (DELETE)” on page 55.)
2. Use UCLIN to remove the MODDEL subentries for the Product A modules from the Product B LMOD entry.
3. Install the new release of Product B.

9.6.5 Cross-Product Load Modules for Products Installed in Different Zones

Cross-product, cross-zone load modules can be created through one of the following methods:

- SMP/E LINK command (done after installation)
- Implicitly defining the modules (done in the JCLIN link-edit step)

Table 12 briefly compares the two methods. It is followed by more information about each method.

Table 12. Comparison of the LINK Command versus Implicit Definition of Modules

Consideration	SMP/E LINK Command	Implicit Definition
Good for products written in high-level languages or that use callable services.		X
Load modules can be automatically serviced.	X	
Modules do not need to be specified individually.		X
SMP/E tracks the cross-zone relationship.	X	

9.6.5.1 SMP/E LINK Command

This method is best when a load module needs to include a few specific modules from another product. To define the cross-zone relationship and create the cross-zone load modules, the LINK command and UCL statements are run by the user. No JCL statements are needed to add the modules to the cross-zone load modules.

If a load module for your product requires a module from another product that is likely to be installed in a different zone, you should describe this in the installation documentation for your product and describe the SMP/E LINK commands the user should run.

When this method is used, SMP/E tracks the cross-zone relationship between the load modules and modules. As a result, cross-zone processing for subsequent APPLY and RESTORE commands can automatically maintain the affected load modules.

The LINK command requires the modules it processes to be stand-alone modules. No assemblies are done by either the LINK command or by cross-zone processing for APPLY and RESTORE commands. Therefore, when packaging a module that you intend to be used as input to the LINK command, make sure it is installed in its target library as either a single-CSECT load module or as part of a totally copied library.

Note: There are times when the LINK command is not appropriate to use--generally, for products that are written in a high-level language and, as a result, include modules from libraries (such as compiler libraries) owned by a different product. In these cases, you should use SYSLIB DD statements to implicitly include the modules. For more information on when to use a

SYSLIB DD statement, see 9.6.5.2, “Implicitly Defining the Modules” on page 103 and the description of the SYSLIB DD statement under 9.6.2, “Link-Edit Control Statements” on page 90.

For an example of using the LINK command, see the *SMP/E User's Guide*. For details on the LINK and UCL commands, see the *IBM SMP/E for z/OS Commands* manual.

9.6.5.2 Implicitly Defining the Modules

This method is best used when a load module must include many modules from other products, and it is difficult and error-prone (and perhaps impossible) to define all the modules to be included.

To implicitly define modules to be included from another product, the SYSLIB statement must be used. (The LIBRARY statement can also be used, if any specific modules should not be included.) Inform the user that the libraries containing the modules must be defined by DDDEF entries in the zone for the product that is including the modules.

Unlike the LINK command, when this method is used, SMP/E does not track the cross-zone relationship between the load modules and modules. However, after a library specified in the SYSLIB DD statement has been updated, the REPORT CALLLIBS command can be used to identify and relink load modules that define a SYSLIB statement. For more information about the REPORT CALLLIBS command, see the *IBM SMP/E for z/OS Commands* manual.

For more information on using the SYSLIB DD statement, see the description of that statement under 9.6.2, “Link-Edit Control Statements” on page 90.

9.6.6 Adding or Changing Load Modules in a PTF

If a PTF needs to add a new load module or change the structure of an existing load module, use the techniques listed below.

- **Adding a new module to an existing load module.** The PTF must ship all of the following:
  - Inline JCLIN describing the new load module structure
  - A ++MOD statement for the new module being added

  When SMP/E installs the PTF, it updates the entries, then performs the link. As a result, the new module is included in the link, and the old load module is replaced.

- **Creating a new load module.** The PTF must ship all of the following:
  - Inline JCLIN describing the new load module structure
  - All of the modules (other than those from other products) that are part of the load module
  - A ++MOD statement for each of those modules (except those from other products)

  When SMP/E installs the PTF, it updates the entries, then performs the link. As a result, the new load module is added to the target library.

- **Deleting a module from a load module (and from a product).** The PTF must ship all of the following:

  – Inline JCLIN describing the new load module structure
  – A ++MOD DELETE statement for the module being deleted

  When SMP/E installs the PTF, it delinks the module from the load module, then removes all references to the deleted module.

- **Deleting a module from a load module (but leaving the module in the product).** The PTF must ship all of the following:
  – Inline JCLIN describing the new load module structure
  – A ++DELETE statement for the load module
  – All the modules (other than those from other products) that are still part of the load module
  – A ++MOD statement for each of those modules (except those from other products)

  When SMP/E installs the PTF, it deletes the load module from the target library, updates the appropriate entries with the new load module definition, and relinks the load module using the modules from the PTF.

Note: In each of these cases, the PTF must contain a ++MOD statement for each module being added or deleted. If the PTF does not contain the ++MOD statement, SMP/E updates the entries but does not invoke the link-edit utility.

9.7 Examples of JCLIN Data

This section shows examples of JCLIN data to define the following:

- Copy, assembler, and link-edit steps for modules
- Copy steps for macros or source
- Assembler steps to create modules from source
- Link-edit steps that use the automatic library call function
- Link-edit steps for load modules residing in a hierarchical file system

9.7.1 JCLIN Data for Modules

The following are some sample job steps for providing SMP/E with the information it needs to copy, assemble, and link-edit modules.

```
/*********************************************************************/
/*                                                                   */
/* Step C1 informs SMP/E that an entire distribution library        */
/* is copied to a target library.  From the INDD operand SMP/E       */
/* determines the ddname of the distribution library (AMACLIB), and  */
/* from the OUTDD operand SMP/E determines the ddname of the target  */
/* library (MACLIB).  SMP/E will use this information to determine   */
/* the target library for subsequent changes that specify an         */
/* element's distribution library as AMACLIB.                        */
/*                                                                   */
/* Although the copy step can be performed using JCLIN, the          */
/* preferred method is to specify the copy in the MCS.               */
/*                                                                   */
/*********************************************************************/
//C1       EXEC PGM=IEBCOPY
//AMACLIB  DD DSN=SYS1.AMACLIB,DISP=SHR
//MACLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//SYSIN DD *
 COPY INDD=AMACLIB,OUTDD=MACLIB  TYPE=MAC

/*********************************************************************/
/*                                                                   */
/* Step C2 shows elements that are selectively copied from a         */
/* distribution library to a target library.  The module name,       */
/* IZZLMODC, is defined by the SELECT MEMBER statement.  The load    */
/* module name, IZZLMODC, is simply the module name.  The INDD       */
/* statement defines the distribution library as AOS14, and the OUTDD */
/* statement defines the target library as LINKLIB.  When the JCLIN  */
/* data is processed, SMP/E sets an indicator (COPY)--the COPY       */
/* indicator means that when the module is link-edited, the link-edit */
/* attributes must be obtained by examining the target library.      */
/*                                                                   */
/*********************************************************************/
//C2       EXEC PGM=IEBCOPY
//AOS14    DD DSN=SYS1.AOS14,DISP=SHR
//LINKLIB  DD DSN=SYS1.LINKLIB,DISP=SHR
//SYSIN DD *
 COPY INDD=AOS14,OUTDD=LINKLIB  TYPE=MOD
 SELECT MEMBER=((IZZLMODC,,R))
/*
```

```
/*******************************************************************/
/*                                                                 */
/* Step A1 defines an assembled module named IZZAMOD1.  The module  */
/* name is specified as the member name on the SYSPUNCH DD statement. */
/*                                                                 */
/* It also defines a macro named IZZAMAC1.  SMP/E will detect the   */
/* invocation of the macro in the assembler SYSIN data.             */
/*                                                                 */
/* NOTE: This method is used to introduce a new element, not to     */
/*       service an existing element.                              */
/*                                                                 */
/*       This example should be used ONLY for supplying inline     */
/*       assembler source, and should NOT be used for elements     */
/*       shipped with ++SRC or ++MOD statements.  The MOD entry     */
/*       resulting from this technique will contain a DISTLIB of   */
/*       SYSPUNCH, which might not be desirable if a ++MOD statement  */
/*       is shipped and the element is installed in a real          */
/*       distribution library.                                     */
/*******************************************************************/
//A1       EXEC PGM=ASMA90
//SYSLIB   DD DSN=SYS1.AMACLIB,DISP=SHR
//SYSPUNCH DD DSN=&&PUNCH(IZZAMOD1),
//            SPACE=(TRK,(1,1,1)),DISP=(,PASS)
//SYSIN DD *
IZZAMOD1 CSECT
         IZZAMAC1  --- INVOKE MACRO
         END IZZAMOD1
/*
```

```
/*********************************************************************/
/*                                                                   */
/* Step L1 shows how to link-edit the previous assembly.  The        */
/* link-edit INCLUDE statement defines module IZZAMOD1.  The module  */
/* name is determined from the member name operand on the INCLUDE    */
/* statement, and the distribution library, SYSPUNCH, is determined  */
/* from the INCLUDE statement's ddname.                              */
/*                                                                   */
/* Step L1 also defines a load module and its target library.        */
/* Load modules IZZLMOD1 is defined by the link-edit NAME            */
/* statement.  The ddname of the target library, LPALIB, is defined  */
/* by SYSLMOD DD statement.  The load module attribute RENT is saved */
/* for use in subsequent link-edits of this load module; the         */
/* parameters LET and LIST are not saved.                            */
/*                                                                   */
/* NOTE: This method is used to introduce a new element, not to      */
/*       service an existing element.                                */
/*                                                                   */
/*       This example should be used ONLY for supplying inline       */
/*       assembler source, and should NOT be used for elements       */
/*       shipped with ++SRC or ++MOD statements.  The MOD entry      */
/*       resulting from this technique will contain a DISTLIB of     */
/*       SYSPUNCH, which might not be desirable if a ++MOD statement */
/*       is shipped and the element is installed in a real           */
/*       distribution library.                                       */
/*                                                                   */
/*********************************************************************/
//L1       EXEC PGM=IEWL,PARM='LET,LIST,NCAL,RENT'
//SYSLMOD  DD DSN=SYS1.LPALIB,DISP=SHR
//SYSPUNCH DD *.A1.SYSPUNCH,DISP=(SHR,PASS)
//SYSLIN   DD *
 INCLUDE SYSPUNCH(IZZAMOD1)
 NAME IZZLMOD1(R)
/*

/*********************************************************************/
/*                                                                   */
/* Step L2 defines two modules and one load module                   */
/* to SMP/E.  Modules IZZAMOD2 and IZZAMOD3 are defined by the       */
/* link-edit INCLUDE statements; the distribution library for each of */
/* these is defined as AOS12.  The load module is defined as         */
/* IZZLMOD2, with LINKLIB as the target library.  The parameters LET */
/* and LIST are not saved.                                           */
/*                                                                   */
/*********************************************************************/
//L2       EXEC PGM=IEWL,PARM='LET,LIST,NCAL'
//SYSLMOD  DD DSN=SYS1.LINKLIB,DISP=SHR
//AOS12    DD DSN=SYS1.AOS12,DISP=(SHR,PASS)
//SYSLIN DD *
 INCLUDE AOS12(IZZAMOD2)
 INCLUDE AOS12(IZZAMOD3)
 ENTRY IZZAMOD2
 NAME IZZLMOD2(R)
/*
```

```
/********************************************************************/
/*                                                                  */
/* Step L3 shows an example of using the OPTIONS option.            */
/* The OPTNAME DD statement allows SMP/E to process the PARM string */
/* even though the options exceed the 100-character limit.          */
/*                                                                  */
/********************************************************************/
//L3       EXEC PGM=IEWBLINK,PARM='OL,AMODE=31,...,OPTIONS(OPTNAME)'
//SYSLMOD  DD DSN=SYS1.LINKLIB,DISP=SHR
//AOS12    DD DSN=SYS1.AOS12,DISP=SHR
//OPTNAME  DD *
 FETCHOPT(PACK,PRIME),RMODE=24
 MAXBLK(256)
/*
//SYSLIN DD *
 INCLUDE AOS12(GIMMPDRV,GIMMPDR1,....)
 ENTRY GIMMPDRV
 SETCODE AC(1)
 NAME GIMMPP(R)
/*

/********************************************************************/
/*                                                                  */
/* Step L4 shows an example of using a SYSLIB concatenation in a    */
/* link-edit step to implicitly include modules from libraries for  */
/* other products.                                                  */
/*                                                                  */
/* Modules MOD00004 and MOD00005 are defined by a link-edit INCLUDE */
/* statement; the distribution library for each of these modules is */
/* defined as AOS12.  The load module is defined as LMOD04, with    */
/* APPLOAD as the target library.  If the CALLLIBS operand is       */
/* specified on the JCLIN command or ++JCLIN MCS, the low-level     */
/* qualifiers of the data sets specified in the SYSLIB concatenation */
/* (PLIBASE and APPBASE) are saved as CALLLIBS subentries in the    */
/* LMOD entry for LMOD04.                                           */
/*                                                                  */
/********************************************************************/
//L4       EXEC PGM=IEWBLINK,PARM='CALL,RENT,REUS'
//SYSLMOD  DD DSN=SYS1.APPLOAD,DISP=SHR
//AOS12    DD DSN=SYS1.AOS12,DISP=SHR
//SYSLIB   DD DSN=SYS1.V2R2M0.PLIBASE,DISP=SHR
//         DD DSN=SYS1.V2R2M0.APPBASE,DISP=SHR
//SYSLIN DD *
 INCLUDE AOS12(MOD00004,MOD00005)
 ENTRY MOD00004
 SETCODE AC(1)
 NAME LMOD04(R)
/*
```

9.7.2 JCLIN Data for Macros and Source

Here is a sample job step for providing SMP/E with the information it needs to copy macros and source:

```
//STEP1    EXEC PGM=IEBCOPY
//AMACLIB  DD DSN=SYS1.AMACLIB,DISP=SHR
//MACLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//AIZZSRC  DD DSN=SYS1.AIZZSRC,DISP=SHR
//IZZSRC   DD DSN=SYS1.IZZSRC,DISP=SHR
//SYSIN    DD *
 COPY INDD=AMACLIB,OUTDD=MACLIB    TYPE=MAC
 S    M=(MAC01,MAC02,MAC03)
 S    M=(MAC11)                    ALIAS OF MAC01
 COPY INDD=AIZZSRC,OUTDD=IZZSRC    TYPE=SRC
 S    M=(SRC04,SRC05)
/*
```

This JCLIN data defines the following to SMP/E:

Element Type	Element Name	Distribution Library	Target Library
Macro	MAC01	AMACLIB	MACLIB
Macro	MAC02	AMACLIB	MACLIB
Macro	MAC03	AMACLIB	MACLIB
Macro	MAC11 (alias)	AMACLIB	MACLIB
Source	SRC04	AJZZSRC	JZZSRC
Source	SRC05	AJZZSRC	JZZSRC

Remember, if the element is fully defined (both the DISTLIB and the SYSLIB are specified on the element MCS), this JCLIN data is not needed.

9.7.3 JCLIN Data for an Assembler Step to Create a Module from Source

Here is a sample job step for providing SMP/E with the information it needs to create a module by assembling source:

```
//STEP1     EXEC PGM=ASMA90
//SYSLIB    DD   DSN=SYS1.MACLIB,DISP=SHR
//SYSPUNCH  DD   DSN=&&PUNCH(SRCA),DISP=SHR
//SYSIN     DD   DSN=SYS1.AIZZSRC(SRCA),DISP=SHR
```

This defines a source module named SRCA, which resides in distribution library AIZZSRC.

9.7.4 JCLIN for Using the Link-Edit Automatic Library Call Function

Starting with SMP/E Release 7, SMP/E provides support for load modules that need to use the link-edit automatic library call function, which enables the load modules to contain modules from multiple products without explicitly specifying those modules on INCLUDE statements in link-edit steps. SMP/E's support for load modules that use the link-edit automatic library call function is called *CALLLIBS support*.

9.7.4.1 Overview of CALLLIBS Support

SMP/E's CALLLIBS support uses the link-edit CALL parameter and a SYSLIB allocation when invoking the link-edit utility to resolve external references in load modules. CALLLIBS support can be useful for a variety of products, including those that:

- Are written in a high-level language and, as a result, include modules from libraries (such as compiler libraries) that are owned by a different product
- Make use of a callable-services interface provided by another product
- Need to include stub routines or interface modules from different products that may reside in other zones

To package a load module that needs to use the automatic library call function, follow these steps:

1. Specify the CALLLIBS operand on the ++JCLIN MCS. CALLLIBS tells SMP/E to:

   - Save the SYSLIB allocation defined by the JCLIN link-edit step in the LMOD entry for the load module. This information is recorded in the new CALLLIBS subentry list.
   - Pass the SYSLIB allocation and the CALL parameter to the link-edit utility for linking the load module.

   Here is an example of the ++JCLIN MCS:

   ```
   ++JCLIN ... CALLLIBS.
   ```

   Note: If CALLLIBS is not specified, the SYSLIB allocation in the link-edit step is ignored and the NCAL parameter is used when invoking the link-edit utility.

2. Provide link-edit JCLIN that defines the SYSLIB allocation for the libraries containing the modules to be implicitly included by the link-edit automatic library call function.

   SMP/E will save the low-level qualifiers of the data sets in the SYSLIB allocation as a CALLLIBS subentry list in the LMOD entry for the load module.

   Here is an example of link-edit JCLIN that defines a SYSLIB allocation for a load module that needs to use the link-edit automatic library call function.

   ```
   //STEP1 EXEC PGM=IEWBLINK,PARM='RENT,REUS'
   //SYSLMOD DD DSN=SYS1.APPLOAD,DISP=OLD
   //AOS12   DD DSN=SYS1.AOS12,DISP=SHR
   //SYSLIB  DD DSN=SYS1.V2R2M0.PLIBASE,DISP=SHR
   //        DD DSN=SYS1.V2R2M0.APPBASE,DISP=SHR
   //SYSLIN  DD *
      INCLUDE AOS12(MOD00001,MOD00002)
      ENTRY MOD00001
      SETCODE AC(1)
      NAME LMOD01(R)
   /*
   ```

3. Inform your users of special requirements for installing the SYSMOD.

   - Users must run SMP/E Release 7 or later to install the SYSMOD.

- Before installing the SYSMOD, users must define DDDEF entries in the target zone that will be used to apply the SYSMOD. DDDEF entries are required for:
  - Each of the data sets in the load module's SYSLIB allocation
  - The SMPLTS data set, which is a new data set introduced in SMP/E Release 7 and is used to link the implicitly-included modules into the load module
- Users who share zones across different releases of SMP/E (that is, run different levels of SMP/E against the same zone) cannot share their zones between SMP/E Release 7 (or later) and previous releases of SMP/E. This is because in SMP/E Release 7, the structure of the LMOD entries has changed to support the new CALLLIBS subentry list. (LMOD entries are typically updated when JCLIN that defined the load module is processed.)

9.7.4.2 Example of a SYSMOD That Implements CALLLIBS Support

The following is a part of a sample function SYSMOD with a load module that needs to use the link-edit automatic library call function. The numbers associate items in the SYSMOD with the steps listed in 9.7.4.1, “Overview of CALLLIBS Support” on page 110.

```
    ++FUNCTION(HXY1100) FILES(3).
    ++VER(Z038).
(1) ++JCLIN CALLLIBS.
    ...
    //STEP1 EXEC PGM=IEWBLINK,PARM='RENT,REUS'
    //SYSLMOD DD DSN=SYS1.APPLOAD,DISP=OLD
    //AOS12   DD DSN=SYS1.AOS12,DISP=SHR
(2) //SYSLIB  DD DSN=SYS1.V2R2M0.PLIBASE,DISP=SHR
    //        DD DSN=SYS1.V2R2M0.APPBASE,DISP=SHR
    //SYSLIN  DD *
       INCLUDE AOS12(MOD00001,MOD00002)
       ENTRY MOD00001
       SETCODE AC(1)
       NAME LMOD01(R)
    /*
    ...
    ++MOD(MOD00001) RELFILE(2) DISTLIB(AOS12).
    ++MOD(MOD00002) RELFILE(2) DISTLIB(AOS12).
    ...
```

The user needs to define DDDEF entries for the data sets specified in the SYSLIB allocation (PLIBASE and APPBASE), as well as for the SMPLTS data set, which SMP/E will use to link-edit the load module. (For details on the SMPLTS data set, see the *IBM SMP/E for z/OS Reference* manual.) Here are examples of defining the DDDEF entries, assuming that the function will be applied to target zone TGT1.

```
(3) SET BDY(TGT1).                 /* Set to target zone.    */
    UCLIN.                         /*                        */
    ADD DDDEF(PLIBASE)             /* Define PLIBASE.        */
        DA(SYS1.V2R2M0.PLIBASE)    /* Data set is cataloged. */
        VOLUME(vvvvvv)             /* Data set VOLUME.       */
        UNIT(SYSALLDA)             /* UNIT MUST be SYSALLDA  */
        SHR.                       /* SHR for read.          */
    ADD DDDEF(APPBASE)             /* Define APPBASE.        */
        DA(SYS1.V2R2M0.APPBASE)    /* Data set is cataloged. */
     VOLUME(vvvvvv)             /* Data set VOLUME.       */
        UNIT(SYSALLDA)             /* UNIT MUST be SYSALLDA  */
        SHR.                       /* SHR for read.          */
    ADD DDDEF(SMPLTS)              /* Define SMPLTS.         */
        DA(SYS1.SMPLTS)            /* Data set is cataloged. */
        SHR.                       /* SHR for read.          */
    ENDUCL.
```

9.7.4.3 Restrictions in CALLLIBS Support

CALLLIBS support puts restrictions on the following:

- **Use of the CALL and NCAL parameters.** Processing of the CALL and NCAL parameters in SMP/E Release 7 is different from processing of those parameters in previous SMP/E releases.

  Before, NCAL was a default parameter passed to the link-edit utility. However, you could use the link-edit UTILITY entry to pass the CALL parameter instead.

  With CALLLIBS support, there is no longer any way to directly tell SMP/E to pass the NCAL or CALL parameter. SMP/E ignores any specification of NCAL or CALL, and instead checks for the CALLLIBS subentry in the load module's LMOD entry to determine which parameter to pass to the link-edit utility when linking the load module.

- **Sharing zones between different releases of SMP/E.** Users cannot share zones between SMP/E Release 7 (or later) and previous releases of SMP/E. This is because in SMP/E Release 7, the structure of the LMOD entries has changed to support the new CALLLIBS subentry list. (LMOD entries are typically updated when JCLIN that defined the load module is processed.)

9.7.5 JCLIN Data for Load Modules Residing in a Hierarchical File System or Java Archive file

A load module can reside in a hierarchical file system (HFS) or Java Archive (JAR) file. To determine where the load module resides, SMP/E uses the following information, in addition to the usual JCL statements needed for load modules:

- The PATH operand on the SYSLIB or SYSLMOD statement associated with the load module. The PATH operand alerts SMP/E to the fact that the load module resides in an HFS or JAR; however, the PATH value specified is ignored.
- The LIBRARYDD comment statement immediately following the statement with the PATH operand. This comment statement specifies the ddname to be associated with the PATH value on the previous DD statement.
- The user-provided DDDEF entry whose name matches the ddname on the LIBRARYDD comment statement. The DDDEF entry specifies the directory portion of the pathname identified by the ddname. SMP/E uses the PATH value specified in the DDDEF entry to allocate the pathname, and does not

check whether this value matches the PATH value specified on the SYSLIB or SYSLMOD DD statement associated with the LIBRARYDD comment.

Following are examples of job steps containing SYSLMOD and SYSLIB DD statements that use the PATH operand.

```
   //STEP1 EXEC PGM=IEWBLINK,PARM='RENT,REUS'
(1)//SYSLMOD DD PATH='/path_name1/'
(2)//*LIBRARYDD=BPXLOAD1
   //AOS12   DD DSN=SYS1.AOS12,DISP=SHR
   //SYSLIN  DD *
      INCLUDE AOS12(MOD00001)
      INCLUDE AOS12(MOD00002)
      ENTRY MOD00001
      NAME LMOD01(R)
   /*
   //STEP2 EXEC PGM=IEWBLINK,PARM='CALL,RENT,REUS'
   //SYSLMOD DD PATH=SYS1.LINKLIB,DISP=OLD
   //AOS12   DD DSN=SYS1.AOS12,DISP=SHR
(3)//SYSLIB  DD PATH='/path_calllib3/'
(4)//*LIBRARYDD=BPXCALL3
(4)//        DD DSN=SYS1.PLIBASE,DISP=SHR
(3)//        DD PATH='/path_calllib4/'
(4)//*LIBRARYDD=BPXCALL4
   //SYSLIN  DD *
      INCLUDE AOS12(MOD00005)
      INCLUDE AOS12(MOD00006)
      ENTRY MOD00005
      NAME LMOD03(R)
   /*
```

(1) Because the SYSLMOD statement specifies a PATH operand, SMP/E expects the next statement to be a LIBRARYDD comment statement.

(2) Using the ddname on the LIBRARYDD comment, SMP/E updates the LMOD entry for LMOD01 to specify a SYSLIB value of BPXLOAD1. The user needs to provide a DDDEF entry for BPXLOAD1, specifying the appropriate pathname.

(3) The SYSLIB DD statement is a concatenation of three DD statements. Two of the DD statements specify the PATH operand.

(4) Using the ddnames on the LIBRARYDD comments and the low-level qualifier of the data set specified on the DSN operand, SMP/E updates the LMOD entry for LMOD03 to specify a CALLLIBS subentry list with the values BPXCALL3, PLIBASE, and BPXCALL4. The user needs to provide DDDEF entries for BPXCALL3 and BPXCALL4, specifying the appropriate pathnames. Likewise, the user needs to define PLIBASE with a DDDEF entry.

Chapter 10. Naming Conventions

This section explains the naming conventions used by the product processes for the following:

- Component identifier (COMP ID)
- SYSMOD IDs
- Element and load module names
- Library names

10.1 Component Identifier (COMP ID)

In the SMP/E environment, code for one product must be uniquely distinguishable from code for other products. The best way to keep your code unique is to start the names of all the elements and load modules for that product with a single unique 3-character identifier. This identifier is called a component code. IBM is offering to register the component codes for your products. The registration ensures that your component code is not used by another products that are registered.

Send a note to ELEMENT@us.ibm.com or ask your IBM representative to contact IBM Poughkeepsie, Department FPLA.

10.2 SYSMOD IDs

The SYSMOD ID of a function is called the function modification identifier (FMID). The FMID is a 7 character identifier that needs to be unique to distinguish one product from another. One way to help ensure this uniqueness is to follow the naming convention tcccrrr, as described below:

- t - is an alphabetic character used to indicate type of function. Avoid the IBM valuse of A,B,C,D,E,F,H and J.
- ccc - is the product version code. You can help guarentee uniqueness by using the the component code.
- rrr - is the release value. This value should be alphanumeric and it should be unique within a product version.

10.3 Element, Alias, and Load Module Names

Element names are assigned to each discrete piece of a product, such as macros, modules, source codes, and panels.

It is important to maintain the uniqueness of element and load module names to ensure that:

- Each element can be identified by its owning product
- Elements are not unintentionally overlayed
- Each element can be serviced correctly

The syntax of element or load module names is either *cccxxxxx* for products with 3-character component codes, or *ccccxxxx* for products with 4-character component

© Copyright IBM Corp. 1986, 2003

codes. The component codes are indicated by *ccc* or *cccc*, and the remaining characters *xxxxx* or *xxxx* are assigned by the product owner.

The first character of the component code generally follows the following conventions to avoid naming conflicts with elements provided by user-written software.

Value **Meaning**

A-I When used by IBM, all three characters of the prefix are generally alphabetic (with some exceptions). Can be used by non-IBM products only if the prefix includes at least one numeric or national character.

J-Z Available for non-IBM products. The prefix can be all alphabetic or can include numeric or national characters.

Note: *ZZZ* is reserved for the first three characters of generic USERMODs written by customers.

Q Used by AS/400

Note: Other operating systems may have different rules for component names.

Packaging Rules (Element and Load Module Names)

- □ Rule 144. Two elements with the same element type cannot have the same name--element names must be unique. This is true regardless of whether the elements are in the same product or in different products. For more information, see 6.1, "General Packaging Rules, Restrictions, and Recommendations for Elements" on page 37.
- □ Rule 145. Load modules must have unique names, which must begin with the product's assigned three-character prefix. However, the same load modules having the same attributes can be defined to two load libraries.
- □ Rule 146. Like-named elements, including aliases, must be in separate target and distribution libraries. These libraries must be in separate RELFILEs. This prevents unintentional overlaying of elements.
  - See 6.2, "Element Ownership" on page 38, 6.1, "General Packaging Rules, Restrictions, and Recommendations for Elements" on page 37 and 10.3.2, "Elements with the Same Name" on page 117 for information about restrictions on like-named elements. See 10.3.3, "Alias Names" on page 117 for information on alias names.
  - See 3.1.1, "Format and Contents of the RELFILE Tape" on page 7 for additional rules and requirements concerning RELFILEs.
- □ Rule 147. If more than one version of a product is intended to coexist in the same zone, the element and load module names must be unique for each version.

When two different elements have the same name and type, the installation process becomes more complicated because each of these elements must be installed in a different zone. You can avoid this predicament by giving each element a unique name, a unique element type, or both.

10.3.1 NLS Considerations for Element Types

Translated elements should use the appropriate data element type, followed by a 3-character national language identifier as a suffix for the element type (for example, ++PNLENU, and ++PNLFRA). Elements that are not translated should not use the national language suffix (for example, ++SKL).

See Table 13 on page 123 for a list of the national language abbreviations, and 6.4, “Data Element Types” on page 39 for a list of the element types.

10.3.2 Elements with the Same Name

Element names must be unique; two elements with the same type cannot have the same name. However, elements that have different types can have the same name provided that they are contained in different FMIDs. For example, ++PNLENU(ABCPANEL) and ++PNLFRA(ABCPANEL) would be valid.

If you need to define elements with the same name (such as HELP) for programming access, you should use HELP as the alias and assign a unique name, in accordance with the corporate naming standard, as the actual element name. For more information, see 10.3.3, “Alias Names” and Chapter 11, “Packaging for National Language Support (NLS)” on page 121.

Note:

10.3.3 Alias Names

Alias names can be assigned to elements or load modules. Alias names are defined on the ALIAS, TALIAS, or MALIAS operand of a ++element statement, or during load module creation. Alias names do not need to begin with the 3- or 4-character component codes. Alias names do not need to be unique within an FMID, but they must be unique within a RELFILE or target or distribution library partitioned data set (PDS). For information about rules for aliases, refer to 10.3, “Element, Alias, and Load Module Names” on page 115.

Note: Macros that are externally invoked can have meaningful alias names; however, the actual name of the macro must conform to the corporate naming standard.

If an alias name is assigned to an element, the RELFILE tape must contain both the element and the alias in a RELFILE. (Alias members must be created using an appropriate utility, such as IEBGENER.)

10.4 Library Names

Whenever possible, elements must be assigned to existing distribution libraries and target libraries (which are specified on the DISTLIB and SYSLIB operands of the element MCS). Otherwise, libraries must follow packaging rules for library names.

Packaging Rules (Library Names)

- ☐ Rule 149. If a target or distribution library is created with a low-level qualifier that has never existed before, that low-level qualifier must be in one of the following formats: *xccczzzz* or *xcccczzz* where:

  - *x* is the letter for a distribution library or a target library.
  - *zzz* or *zzzz* is whatever the product developer chooses to use to keep the name unique.

  Exception: A data set name need not conform to this format if *all* of the following are true:

  1. The data set name is required to have a non-conforming low-level qualifier for unavoidable technical reasons. (For example, C language header file data sets are required by the C compiler to use the low-level qualifier of "H".) Your packaging representative must agree with this assessment before you conclude that it applies to your product.
  2. The data set is *not* specified as a target library in any JCLIN data, either on a SYSLMOD DD statement or on an EXEC statement.
  3. The data set name is *not* specified in a SYSLIB concatenation in any JCLIN data.

  Data sets that qualify under this exception must still use ddnames with the format *xccczzzz* or *xcccczzz* as defined above, to comply with rule 149.1.

- ☐ Rule 149.1. Every target and distribution library must have a unique ddname.

- ☐ Rule 150.1. If a data set whose name does not use the *xccczzzz* format is renamed for any reason, it becomes a "new" data set, and must comply with all Packaging Rules associated with new data set names.

  Exception: A data set name need not conform to this format if *all* of the following are true:

  1. The data set name is required to have a non-conforming low-level qualifier for unavoidable technical reasons. (For example, C language header file data sets are required by the C compiler to use the low-level qualifier of "H".) Your packaging representative must agree with this assessment before you conclude that it applies to your product.
  2. The data set is *not* specified as a target library in any JCLIN data, either on a SYSLMOD DD statement or on an EXEC statement.
  3. The data set name is *not* specified in a SYSLIB concatenation in any JCLIN data.

  Data sets that qualify under this exception must still use ddnames with the format *xccczzzz* or *xcccczzz* as defined above, to comply with rule 149.1.

- ☐ Rule 151. A product's execution must not depend on the high-level qualifier of any data set names. Product code should refer only to ddnames.

  Note: This rule does not apply to modules executing as *nucleus initialization programs (NIPs)* RIMs, which must specifically refer to SYS1 libraries.

Packaging Rules (Library Names)

- ☐ Rule 151.1. Target and distribution libraries provided by products must not require hard-coded high-level qualifiers. Documentation and sample jobs referring to these libraries should explain that high-level qualifiers can be modified by the installer.
- ☐ Rule 151.2. The following applies to any PTF that requires the creation of a new target or distribution library:
  1. The PTF must have a ++HOLD for ACTION, and explain how to allocate the new library and how to create the DDDEF entries.
  2. The new library must conform with all z/OS packaging rules, including naming rules.
  3. The PTF must replace the product's allocation and DDDEF jobs to reflect these updates, so that subsequent service updates (SUPs) of the product will contain correct information.

A process that depends on a specific data set name may restrict customer processes or naming conventions. You should design your product to rely only on specific ddnames.

Using the low-level qualifier as the ddname for a data set ensures that the ddname will be unique in the SMP/E zone. An advantage of a unique ddname is that the SMP/E DDDEF for that data set is also guaranteed to be unique. If either the ddname or the DDDEF is not unique, products might unnecessarily prevent other products from being installed in the same zone.

Packaging Recommendations

The variable portion of the library name should be used to describe the library. For example, the type of elements found in the library could be indicated by MOD, MAC, or PNL, or the national language of the library could be indicated by identifiers such as ENU, FRA, or ESP. Table 13 on page 123 lists the national language identifiers.

End of Packaging Recommendations

Chapter 11. Packaging for National Language Support (NLS)

There are packaging rules and considerations for products that have elements that require translation for *national language support (NLS).* This section shows several variations of base and additive dependent function SYSMODs and how they are packaged with their language-support dependent function SYSMODs.

Notes:

1. For more information on NLS, see the following:
   - *National Language Information and Design Guide*, SBOF-3101, Series of Books
2. Refer to 7.2.4, "Deleting SYSMODs (DELETE)" on page 55 for information about language-support dependent functions not deleting additive dependent functions.

Packaging Rules (Language-Sensitive Elements)

- ☐ Rule 154. Each supported language must be individually orderable by its own media feature code. The feature code must ship everything needed to install the function and the language, including all required functions and installation publications.

Packaging Recommendations

Languages can be packaged in a number of ways, including:

- Each language has a separate FMID
- One language is included in the base function and the rest have separate FMIDs
- All languages are packaged in the base FMID

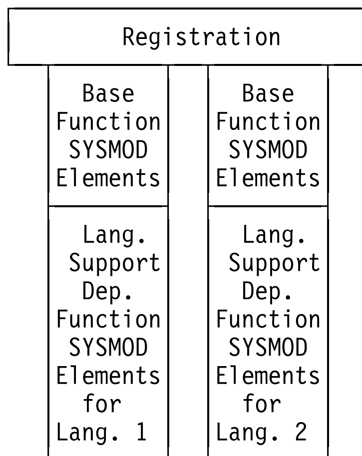
The decision should be based on such factors as:

- If the language functions are large, separate FMIDs permit customers to save space by only installing the languages they wish to use
- If most customers will want most or all of the languages, using a single FMID makes installation easier without wasting space
- How many tapes will be required to ship the various combinations of functions?

End of Packaging Recommendations

Following is an overview of how to package NLS support for a single base function.

© Copyright IBM Corp. 1986, 2003



In this example, the elements that do not have to be translated are in a single base function SYSMOD, and the translated elements for each language are in a separate language-support dependent function SYSMOD, one for each language. For each supported language, the base function SYSMOD is packaged with the appropriate language-support dependent function SYSMOD.

Thus, two media feature codes are required:

- One for the Language 1 environment
- One for the Language 2 environment

For more detailed examples of packaging language-sensitive elements, see Chapter 13, "SYSMOD Packaging Examples" on page 135 and 13.4, "Example 3: Dependent Functions" on page 144.

11.1 Element Types for Translated Data Elements

Packaging Rules (Language Abbreviations)

- ☐ Rule 155. When the data element MCS indicates the language being supported, use one of the national language identifiers shown in Table 13 on page 123 as the three-character suffix for the element type.
- ☐ Rule 156. Each language variant of a data element type constitutes a distinct element type, and rules applying to element types apply to every such variant. For example, ++PNLENU and ++PNLDEU are two different element types.

You should not use a national language identifier for a data element that was not translated.

Table 13. National Language Identifiers Used for Language-Unique Elements. *See 6.4, "Data Element Types" on page 39 for a list of element MCS you can use these identifiers with.*

Value	Language	Value	Language
ARA	Arabic	HEB	Hebrew
CHS	Simplified Chinese	ISL	Icelandic
CHT	Traditional Chinese	ITA	Italian (Italy)
DAN	Danish	ITS	Italian (Switzerland)
DES	German (Switzerland)	JPN	Japanese
DEU	German (Germany)	KOR	Korean
ELL	Greek	NLB	Dutch (Belgium)
ENG	English (United Kingdom)	NLD	Dutch (Netherlands)
ENP	Uppercase English	NOR	Norwegian
ENU	English (United States)	PTB	Portuguese (Brazil)
ESP	Spanish	PTG	Portuguese (Portugal)
FIN	Finnish	RMS	Rhaeto-Romanic
FRA	French (France)	RUS	Russian
FRB	French (Belgium)	SVE	Swedish
FRC	French (Canada)	THA	Thai
FRS	French (Switzerland)	TRK	Turkish

Table 13 might not reflect the most currently supported values. For the latest information on national language identifiers, see the *SMP/E Reference* manual.

11.2 Planning the Physical Media for NLV

There are a number of alternatives for packaging your *national language version (NLV)*. You may want to consider separating your code into code that can be translated and code that won't be translated by FMID(s).

If your product does not translate any elements and has no future plans to translate, it is not required to separate into separate FMIDs. If the translation plans for the product change at some future time, then the code must be separated so that the language features can be packaged. Therefore, you should consider the possibility in the initial design of the product and its packaging so that future disruption can be avoided.

Chapter 12. Packaging for Special Situations

This chapter offers packaging suggestions to accommodate the following:

- High-level languages (HLL)
- C language prelinker
- Workstation code

12.1 High-Level Languages

Because SMP/E supports the automatic library call facility through the use of SYSLIB DD statements, products now have an alternative to postinstallation link-edit jobs or explicitly defining all the modules to be included in load modules. This section contains two parts: one for packagers who can take advantage of the support in SMP/E and one for those who cannot.

Packaging Recommendations

- $PRIVATE or $PRIVnnnn symbols should not appear in load modules. Products written in high-level languages should use statements that will assign names to all code and data sections.
- A new Version or Release of a product that requires an HLL for installation (C/C++, COBOL, PL/I, FORTRAN) should use LE and only LE.

End of Packaging Recommendations

12.1.1 Support in SMP/E Release 7 and Later for the Automatic Library Call Facility

If you require SMP/E Release 7 or later as the minimum level of SMP/E for installing your product, you can use SYSLIB DD statements and the automatic library call facility to implicitly include modules. For more information about using the SYSLIB DD statement for such products, see the description of that DD statement in 9.6.2, “Link-Edit Control Statements” on page 90.

Packaging Rules (Libraries)
☐ Rule 158.1. If a ++PROGRAM element is pre-bound with parts from another product, it must require the other product as a functional (non-installation) requisite. This will avoid problems in customer environments with varying levels of the product.

12.1.2 If You Cannot Use the Automatic Library Call Facility

SMP/E Release 7 (and earlier) provides limited support of high-level languages. It expects to explicitly know all the components of a load module. SMP/E does not exploit the automatic library call option when the link-edit utility is invoked; however, this automatic library call option is used by most high-level languages to include the resident library routines in the load module.

There are two ways to address this problem:

© Copyright IBM Corp. 1986, 2003

- Use a postinstallation link-edit job. This is the most flexible method.
- Use JCLIN to explicitly identify all the library routines to SMP/E.

Each of these options requires extra packaging and installation steps. There is no complete or easy solution.

12.1.2.1 Using a Postinstallation Link-Edit Job

For this approach, you do not identify the resident libraries to SMP/E. The typical SMP/E link-edit options are NCAL and LET. The link-edit utility issues messages indicating unresolved external references, but these can be ignored. You can provide the user with a link-edit automatic library call option job to be run after installation in order to include the required library routines.

Considerations

1. This requires an additional, manual step to complete the installation. In such cases, errors are likely.
2. SMP/E will not know about this extra link-edit and the resident libraries routines that now are in the load module. This may not be a problem, because SMP/E always includes the old load module when creating a new one. In this case, as long as no changes are required to the resident library routines, those routines continue to exist in the new load module as before.
3. SMP/E does not relink the load module to incorporate maintenance or product-level changes processed for the resident libraries.
4. It is difficult to install code changes affecting the resident libraries. One approach is to rerun the postinstallation job. This is a manual process, and PTFs requiring it need to be held for an ACTION reason ID. Another disadvantage is that the input load module already contains all the resident library routines previously added (from the prior invocations of the job). The automatic library call option works only for unresolved references left *after* the inclusion of the existing load module; only net additions can be processed. Changes or deletions are not done unless you can return to the original point of including only your pieces in the link-edit. Similarly, you cannot use the postinstallation link-edit to bring maintenance to current levels for the resident libraries, because the existing versions are always included first.

There are some alternatives to deal with the problems described in items 3 and 4. You can do a postinstallation link-edit with only the modules that the product owns. The automatic library call option would then include the resident library routines. Do one of the following:

- Code the postinstallation link-edit job to include from the distribution libraries rather than the target load module. This forces complete processing of the automatic library call option, because the distribution library data sets have only your product's code in them.

  These are some of the problems with this method:

  – The JCL is more complex. You may have to include multiple parts instead of just including the target load module.

  – The user must accept the product into the distribution libraries before running the link-edit job.

- Because the user must accept all required maintenance before rerunning postinstallation link-edit, the SMP/E RESTORE command cannot be used for recovery.

- Provide two copies of each affected load module: one to be the target of the postinstallation link-edit, and one to be the source.

  The load modules would be the same from an SMP/E standpoint and would have the same contents. They could be defined to SMP/E as either two load modules in the same library with different names or as the same load module in two different target libraries. SMP/E can handle either case. A consideration is the amount of space already used in the target library. If the number of load modules is small, you may opt to have two copies in the same target library: one executable, and one not. The alternative creates additional complexity by using a new target library.

In either case, the postinstallation link-edit must be set up to include the second load module to relink the first (executable) load module. In this way, whenever PTFs for your product are installed, SMP/E automatically includes the changes in both load modules. If a PTF does not change the resident libraries, nothing needs to be done. However, if a PTF does change the resident libraries, a hold reason ID of ACTION can be specified for the PTF to indicate that a postinstallation link-edit job needs to be rerun; the automatic library call option includes the correct set of resident libraries. Because all of this is done against target libraries, the RESTORE command can be used to remove bad maintenance and can be followed by the link-edit job again, if necessary.

12.1.2.2 Using JCLIN to Identify Library Routines

You can define to SMP/E all the library routines used by the link-edit utility. You must use JCLIN to identify all the INCLUDE statements for the library routines needed.

If all the following conditions are true, SMP/E can correctly construct the load module without using the automatic library call option.

- The target load module is *new* (not preexisting).
- The required resident library routine is installed in the *same* target/DLIB set of SMP/E zones as that used for the new product.
- You have supplied correct JCLIN describing the routines needed. In this case, SMP/E uses its load module build function to generate the correct INCLUDEs in the link-edit input to build the load module.

Subsequent PTF maintenance to either your product or to the resident library routines causes the load module to be relinked with the updated parts, because SMP/E knows where to find all the load modules that must include the part. This link-edit process includes the *old* load module when the *new* load module is created; SMP/E ensures that no parts are lost.

Considerations

1. The JCLIN must be accurate so that no required resident routines are missed.
2. The load module must be *new*. If the load module is already known to SMP/E, SMP/E just includes the old copy without invoking load module build.

3. Changes in the required resident routines are difficult. For example, what if you change the source code such that a different library routine is required? If nothing else is done, SMP/E uses the JCLIN as you supply it; however, the load module build function is not called, because the load module already exists. Additional routines are not included as required.
4. Installation of new releases also have complications for similar reasons. Typically, the new release uses SMP/E ++VER DELETE processing to remove the old level. SMP/E tries to delete the load modules owned by the old FMID before applying the new release. However, because SMP/E knows that the load modules contain pieces belonging to other FMIDs (the resident library routines), it does not do a total delete. SMP/E deletes the old pieces but leaves the load module in place with the associated resident library routines still there.

   When the new product is installed, SMP/E knows that the load module still exists, so load module build is not used, and SMP/E includes the old load module. This might not cause a problem if the exact same set of resident routines is required. If there are any changes to the resident routines, however, the load module will not be correct.
5. There can be drawbacks to automatically relinking a product's load modules whenever there is a maintenance or product-level change to the resident libraries:
   - There may be a problem in the new level of subroutine because of code problems and interface changes. This can cause problems, even though you did not change anything.
   - If there is a product change, the situation is worse if the new level of the resident library deletes the old level.

     In this case, SMP/E does the following:

     a. Deletes the old pieces of the resident library wherever they occur. This means it removes them from the load modules.
     b. Deletes the SMP/E information about the old pieces from the SMP/E zone. This includes deleting the links to your load modules.
     c. Installs the new pieces of the resident library. The load modules are not updated with the equivalent new parts.

     If you are using SMP/E Release 7 or later, however, SMP/E maintains a record of any modules from a deleted product that were included in a load module of another product. If the deleted modules are reintroduced, SMP/E automatically link-edits the load module to include the borrowed modules. This can be helpful but, depending on the products involved, SMP/E may try to include modules that no longer exist, and it might not include all the modules you need.

12.2 Using the C Language Prelinker

SMP/E does not invoke the C Prelinker. The C Prelinker is needed for:

- Reentrancy

  Some products have avoided use of the C Prelinker by writing the code in a naturally reentrant format.

Note: If your product has already been developed, this option may not work for you.

- Support of long names

There are instances where using the C Prelinker cannot be avoided. The following example explains how a product can avoid a packaging problem if the C Prelinker must be used.

12.2.1 Example of a Product Requiring the C Prelinker

Product A, which is written in C, includes:

- **Load module ABCLMOD**, which contains these CSECTs used as input to the Prelinker (shown in Figure 5 on page 130):
  - **CSECT ABCM1**
  - **CSECT ABCM2**
  - **CSECT ABCM3**
- **Text deck ABCT1**, which is the Prelinker output from ABCM1, ABCM2, and ABCM3. ABCT1 is shipped as a module in product A.

The MCS that describes module ABCT1 is:

```
++MOD(ABCT1) CSECT(ABCM1,ABCM2,ABCM3) DISTLIB(nnnnnnn) RELFILE(n).
```

The product tape for product A contains module ABCT1, which must be in link-edit format, as required for modules on RELFILE tapes. You do not need MCS for CSECTs ABCM1, ABCM2, or ABCM3. You also need to provide JCLIN to indicate that load module ABCLMOD contains module ABCT1.

Suppose that an error is later discovered in CSECT ABCM1. The service process supplies an updated copy of CSECT ABCM1, in addition to CSECTs ABCM2 and ABCM3, which are at the same level as were shipped on the product tape. All three CSECTs must be shipped so that an updated text deck ABCT1 can be created. Module ABCT1 is shipped as an inline module replacement PTF. (It is a service requirement for PTFs that modules be in inline format.) The MCS for this new level of ABCT1 is:

```
++MOD(ABCT1) CSECT(ABCM1,ABCM2,ABCM3) DISTLIB(nnnnnnn).
```

When the PTF is installed, SMP/E invokes the link-edit utility to link-edit module ABCT1 into load module ABCLMOD. Because the structure of load module ABCLMOD has not changed, no JCLIN is required.

Servicing modules shipped in this manner can have some complications. Large PTFs may result because both updated CSECTs and unchanged CSECTs for a module must be shipped when servicing that module. The advantage of this method is that the product can use the normal service process, and no special customer action is required when installing service.

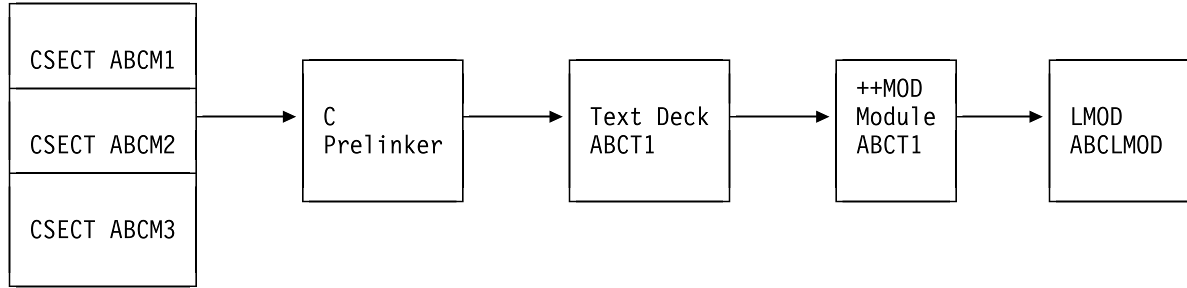


Figure 5. Using the C Prelinker to Create Load Module ABCLMOD

12.3 Packaging Workstation Code to Be Installed on the Host

There may be instances where workstation code needs to be installed on the host and downloaded to workstations. One of the advantages of delivering workstation code to an z/OS host is that it can be maintained under SMP/E control; central service can be used to supply updates. Because such code is z/OS installable, it must also comply with the z/OS packaging rules.

Packaging Rules (HFS and JAR)

- □ Rule 189. All products installing into the HFS or JAR should install their product code under /usr/lpp/zzzzzzzz/, (zzzzzzzz - company name) and (if necessary) instruct the customer to place customization data under /etc/xxxxxxxx, where xxxxxxxx is the product or element's choice, and can be any length or any number of directories.

  Products should register their zzzzzzz value(s) and any symbolic links they create with the Packaging Rules Owner. Two products may not use the same value; if a new product tries to register a value already owned by an existing product, the new product will be required to choose a different value.

  Products must not install into HFS or JAR paths owned by other products without the permission of the owning product.

- □ Rule 189.1. HFS or JAR paths with the name format /usr/lpp/zzzzzzzz/ should be used as target libraries; any path created for another purpose must use a different naming convention.

Packaging Rules (HFS and JAR)

- Rule189.2. z/OS non-exclusive elements/products must provide a directory that contains the version/release identifier for that product or element. The format should be /usr/lpp/xxx/VnRnMn (capital VRM) unless the product has a need for another format. In addition, when these non-exclusive elements/products provide sample jobs to create a filesystem (HFS data set) that will hold their product code, they must have the version/release modifier directory be the mountpoint and not the product name directory. For example /usr/lpp/xxx/VnRnMn should be used as the mountpoint instead of /usr/lpp/xxx.

  By not having a release identifier as a directory, customers will not be able to install multiple releases of the product on the same system. This is needed to allow for testing of new releases on the same system that may have a previous release of the product, currently used in production. For exclusive elements, there is no need for a versions/release directory, since there will only be one copy of that element for that release of the operating system. By using the version/release directory as the mountpoint, one can easily transport a particular level of a product from a build type system to a production center, without having to do a full install as is commonly done today by customers.
- Rule 190. Thus rule has been deleted.
- Rule 191. Symbolic links must not exist in the /tmp, /dev, /var or /etc directories, or in any directory under them, that point into a different directory or file.
- Rule 191.1. All symbolic links, no matter how they are created, must be relative, not absolute.
- Rule 191.2. z/OS non-exclusive elements/products must not create symbolic links outside their directory structure (ie. /usr/lpp/xxxxxx) to the root directory. Instead the element should instruct the customer to add necessary directories to environment variables such as PATH, LIBPATH, CLASSPATH, NLSPATH etc...

  By not creating symbolic links, both internal and external customers can have multiple releases of the product on a system. This will allow for flexibility when testing out new levels and upgrading to the next release. It will also allow for transporting the product files from one system to another, without a formal install. When a symbolic link is created from say /usr/lpp/prod1/bin/utilityA to /bin/utilityA, you have directly tied it to one release level. For exclusive elements, the method of creating symbolic links should be continued to be used.
- Rule 192. Products must not install anything directly into the /etc directory during APPLY processing; the /etc directory is used only for customization data. Shell scripts invoked by SMP/E must not install or change files in the /etc directory.
- Rule 193. Permission bits for every directory in the HFS or JAR must be User greater than or equal to 7, Group greater than or equal to 5, and Other greater than or equal to 0. Likewise, permission bits for every file in the HFS or JAR must be User greater than or equal to 6, Group greater than or equal to 4, and Other greater than or equal to 0.

Packaging Rules (HFS and JAR)

- ☐ Rule 194. If Product A's SMPMCS indicates a shell script supplied by Product B in a SHSCRIPT operand, A must PRE or REQ B.
- ☐ Rule 195. Products must not require the installer to enter UNIX System Services shell line commands; the product must provide jobs or scripts instead.

  Exception: If the Program Directory documents the option of creating a new HFS or JAR for the product, and a MOUNT command is documented as part of this option, the MOUNT command is not required to be in an EXEC.
- ☐ Rule 196. Shell Scripts invoked by SMP/E during DELETE processing must clean up whatever they did during APPLY processing.
- ☐ Rule 196.1. A product's installation must not require the editing of any shell scripts invoked by SMP/E.
- ☐ Rule 199. The only UNIX ID allowed to be specified in the PARM operand of the ++HFS or ++JAR statement in the SMPMCS, or in the PARM operand of the EXEC statement in JCLIN, is a UID of 0. If a different value is required, a User ID or Group ID name must be specified, and must be documented in the Program Directory. If the installation may require the name to be changed, the Program Directory must indicate that a postinstall task is required.

Packaging Recommendations

- The permission bits for HFS or JAR files should be User=7, Group=5, Other=5 for executables, and User=6, Group=4, Other=4 for all other files. (NOTE: there may be some exceptions for daemons, started tasks, and other setuid 0 programs.)
- The permission bits for HFS and JAR directories should be User=7, Group=5, Other=5.
- Products should not require a product-specific HFS or JAR. Instead, document the amount of space needed for the product, and allow the installer to choose whether to install in the root HFS or JAR.
- The MKDIR EXEC shipped with the product should only create paths under /usr/lpp/. If paths must be created in other directories, it should be done in a customization EXEC, or dynamically by the product during execution.
- In link edit JCLIN, the PATH= value on the //SYSLMOD DD statement should match the path name in the DDDEF entry indicated by the subsequent LIBRARYDD comment.
- If an HFS or JARpath needs a SYMLINK that does not conform to the HFS and JAR path naming rule (/usr/lpp/xxxxxxxx), a Rules-compliant SYMLINK should be added in the SMPMCS or MKDIR EXEC, and then changed in an optional customization job.

- For symlinks that pointed to a different target in a previous MKDIR, the unlink should first check for the known target if SMP/E creates a new symlink with the same name.
- For symlinks replaced by files, use an unlink that allows a file to exist where a symlink used to be.
- Elements containing NLV message catalog parts are recommended to install into the following directories:
  - For English - /usr/lpp/prodA/nls/msg/C/zzzzzzzz/
  - For Japanese - /usr/lpp/prodA/nls/msg/Ja_JP/zzzzzzzz/
  - For Simplified Chinese - /usr/lpp/prodA/nls/msg/Zn_CN/zzzzzzzz/
  - ..and so on

End of Packaging Recommendations

Chapter 13. SYSMOD Packaging Examples

This chapter illustrates relationships when developing and servicing the following sample products:

- Product A: defining a stand-alone base function (with support for only U.S. English elements)
- Products B and C: defining corequisite base functions (with support for only U.S. English elements)
- Products B and C: defining dependent functions (with support for only U.S. English elements)
- Products C, D, and E: defining base functions with prerequisites (with support for only U.S. English elements)
- Product E: defining mutually exclusive dependent functions (with support for only U.S. English elements)
- Products X and Y: defining functions that support more than one language
- Products K, L, and M: changing the contents of products

13.1 Conventions Used in This Chapter

Please Note

This chapter contains "skeleton" SYSMOD packaging examples to highlight packaging concepts. The syntax used in the examples is not complete. For example, certain operands such as DISTLIB and SYSLIB are not shown, to focus on other operands whose use is being demonstrated. Therefore, before using these examples, contact your packaging representative for more information.

To make the SYSMOD packaging examples easier to read, shortened forms of product names and SYSMOD IDs are used. For example:

Product A single letter is used, such as Product A or Product B.

Function Each base function FMID starts with "H", and each dependent function FMID starts with "J". This is followed by the product letter and a number (for example, Function HA0 or Function JB1). (For language-support dependent functions, an abbreviation indicating the language is also included--for example, JA0ENU.)

Program temporary fix, or PTF The letter P, a letter associated with the product, and a single number are used, such as PTF(PA1) or PTF(PA2).

APAR fix The letter R, a letter associated with the product, and a single number are used, such as APAR(RA1).

SYSMOD in general A single number is used, such as SYSMOD(1) or SYSMOD(2).

© Copyright IBM Corp. 1986, 2003

Element A single letter is used, such as MOD(J) or MOD(K). (For language-sensitive elements, an abbreviation identifying the product is also included--for example, AP to identify an element for product A.)

13.2 Example 1: A Stand-Alone Function

Suppose you have developed a new product (J) that has no dependencies on other SYSMODs that may be installed on the same system, and that supports only U.S. English in its dialog panels and messages. These are some of the SYSMODs you may define in the course of developing and servicing product A:

- The initial release of the product. This release consists of the:
  - Base function
  - Language-support dependent function for English
- PTF service for the initial release
- PTF service that depends on previous service
- Replacing the initial release
- Ensuring that a fix for a previous release is not lost
- Integrating PTF service for a service update

13.2.1 Initial Release

The first release of product A is packaged as FMID HA0 (a base function) and FMID JA0ENU (its language-support dependent function). Because function HA0 is the base function, it is an **unconditional prerequisite** for function JA0ENU. This relationship is defined by the FMID operand on the ++VER statement for JA0ENU, as shown in Figure 6. However, because these functions have no relationships with other SYSMODs, no other requisites need to be specified on their ++VER statements.

```
Product A

++FUNCTION(WA0)
  REWORK(2003060).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).

++FUNCTION(XA0ENU)
  REWORK(2003060).
++VER(Z038) FMID(WA0).
++JCLIN RELFILE(1).
++PNLENU(AP).
++MSGENU(AM).
```

Figure 6. Initial Release

13.2.2 PTF Service for the Initial Release

Suppose a user has reported an error in function HA0, and you have packaged the fix as an APAR (RA1) to correct the problem on that user's system. Now it appears that the problem may occur on all users' systems. To distribute the fix as service to all users, you do additional testing on the correction and package it as PTF(PA1). The fix is applicable to function HA0, so HA0 is an **unconditional prerequisite** for PTF(PA1). You define this relationship by coding a ++VER statement that specifies function HA0 as the FMID, as shown in Figure 7. To ensure that the APAR cannot be installed on top of the PTF, and thus regress the changes, you should also have the PTF **supersede** the APAR.

```
Product A

++FUNCTION(WA0)
  REWORK(2003060).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).

++APAR(RA1).
++VER(Z038) FMID(WA0).
++MOD(J).

++PTF(PA1).
++VER(Z038) FMID(WA0)
            SUP(RA1).
++MOD(J).
```

Figure 7. PTF Service for the Initial Release

13.2.3 PTF Service That Depends on Previous Service

After a while you have some service for function HA0. This fix depends on some of the changes made by PTF(PA1). You package the fix as a PTF, PA2, for product A. The fix is applicable to function HA0, so HA0 is an **unconditional prerequisite** for PTF(PA2). You define this relationship by coding a ++VER statement that specifies function HA0 as the FMID, as shown in Figure 8 on page 138. In addition, because PA2 depends on changes made by PA1, PA1 is also an **unconditional prerequisite** for PTF(PA2). You define this relationship by coding a ++VER statement that specifies PTF(PA1) as a prerequisite.

Product A
++FUNCTION(WA0)   REWORK(2003060). ++VER(Z038). ++JCLIN RELFILE(1). ++MOD(J). ++MOD(K).
++APAR(RA1). ++VER(Z038) FMID(WA0). ++MOD(J).
++PTF(PA1). ++VER(Z038) FMID(WA0)   SUP(RA1). ++MOD(J).
++PTF(PA2). ++VER(Z038) **FMID(WA0)**   **PRE(PA1).** ++MOD(K).

Figure 8. PTF Service That Depends on Previous Service

Note: To make the rest of the examples in this section easier to read, none of them show APAR fixes being superseded by PTFs. (APAR fixes are normally superseded by the PTFs that include them.)

13.2.4 Ensuring That a Fix for a Previous Release Is Not Lost

Suppose you are completing base function HA2, the second release of product A, and have included all the PTFs that were issued for HA1. After doing this, however, you had to add another PTF(PA8) to fix module B in function HA1. But, because the development cycle for function HA2 has passed the APAR cutoff point, the fix cannot be included in HA2. You want to make sure that any users who install PTF(PA8) do not lose those corrections when they install function HA2.

To do this, you must also code PTF(PA9), which fixes the same problem as PA8, only for function HA2. You must ensure that users who had installed PA8 will install PA9 along with function HA2. PTF(PA9) is, therefore, a **conditional corequisite** for function HA2. You define this relationship by coding an ++IF statement in PTF(PA8), as shown in Figure 9 on page 139.

Product A	
Old Release	New Release

++FUNCTION(WA1)
  REWORK(2003050).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).
	++FUNCTION(WA2)
	  REWORK(2003100).
	++VER(Z038)
	  DELETE(WA1)
	  SUP(WA1).
	++JCLIN RELFILE(1).
	++MOD(J).
	++MOD(K).

++PTF(PA8). ++VER(Z038) FMID(WA1). **++IF FMID(WA2) REQ(PA9).** ++MOD(K).	**++PTF(PA9).** ++VER(Z038) FMID(WA2). ++MOD(K).

Figure 9. Ensuring That a Fix for a Previous Release Is Not Lost

When a user tries to install PTF(PA8), SMP/E does one of two things:

- If function HA2 is already installed, SMP/E cannot install PTF(PA8) and does not know that PA9 is required for HA2. Because PA8 was never installed on function HA1, the system is not at a lower level when function HA2 is installed without PA9. PA9 is eventually installed when the user processes service for HA2, and the problem is fixed.
- If function HA2 is not yet installed, SMP/E notes that PA9 is needed for HA2, and saves this information. Later, if the user tries to install function HA2, SMP/E makes sure PTF(PA9) is also installed. This ensures that the corrections from PA8 are not lost.

13.2.5 Replacing the Initial Release

Suppose there are a number of improvements you want to make in product A, so you are thinking of packaging a new release. This new release could delete the initial release, supersede it, or both. (See Table 5 on page 21 if you need to review the differences.)

In this example, base function HA1 supersedes and deletes base function HA0. Function HA1 is, therefore, an unconditional replacement for HA0. You define this relationship by coding the SUP and DELETE operands on the ++VER statement for function HA1, as shown in Figure 10 on page 140.

A new release of the language-support dependent function is required for function HA1. This new release (JA1ENU) must supersede the previous release (JA0ENU), as shown in Figure 10 on page 140.

Note: There is no need for HA1 or JA1ENU to delete JA0ENU, because JA0ENU is automatically deleted when HA0 is deleted. However, an explicit deletion is recommended for purposes of documentation.

<table>
<tr><th colspan="2">Product A</th></tr>
<tr><th>Old Release</th><th>New Release</th></tr>
<tr><td><pre>++FUNCTION(WA0)
  REWORK(2003060).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).</pre></td><td><pre>++FUNCTION(WA1)
  REWORK(2003250).
++VER(Z038)
  DELETE(WA0)
  SUP(WA0).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).</pre></td></tr>
<tr><td><pre>++FUNCTION(XA0ENU)
  REWORK(2003060).
++VER(Z038) FMID(WA0).
++JCLIN RELFILE(1).
++PNLENU(AP).
++MSGENU(AM).</pre></td><td><pre>++FUNCTION(XA1ENU)
  REWORK(2003250).
++VER(Z038) FMID(WA1)
            SUP(XA0ENU).
++JCLIN RELFILE(1).
++PNLENU(AP).
++MSGENU(AM).</pre></td></tr>
</table>

Figure 10. Replacing the Initial Release

Suppose the previous example left the ++IF statement out of PTF(PA8). This creates the possibility of customers regressing their systems by installing HA2 without PTF(P9).

To avoid this problem, the product owner could change the packaging for HA2 and define PA9 as an **unconditional requisite**. This is done by specifying the REQ operand on the ++VER statement, as shown in Figure 11.

<table>
<tr><th colspan="2">Product A</th></tr>
<tr><th>Old Release</th><th>New Release</th></tr>
<tr><td><pre>++FUNCTION(WA1)
  REWORK(2003050).
++VER(Z038)
  DELETE(WA0)
  SUP(WA0).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).</pre></td><td><pre>++FUNCTION(WA2)
  REWORK(2003100).
++VER(Z038)
  DELETE(WA0,WA1)
  SUP(WA0,WA1).
  REQ(PA9).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).</pre></td></tr>
<tr><td><pre>++PTF(PA8).
++VER(Z038) FMID(WA1).
++MOD(K).</pre></td><td><pre>++PTF(PA9).
++VER(Z038) FMID(WA2).
++MOD(K).</pre></td></tr>
</table>

Figure 11. Correcting an Erroneous Post-Cutoff PTF

13.3 Example 2: Corequisite Base Functions

Suppose you have developed a new function that involves elements from two different products, B and C. Each product provides specific aspects of the function, but the code works properly only if the two products are installed together. Both products support only U.S. English in their dialog panels and messages. These are some of the SYSMODs you might have to define in the course of developing and servicing products B and C:

- The initial releases of the products. These consist of:
  - The base functions
  - The language-support dependent functions
- PTF service for one of the base functions
- Cross-product service between the base functions
- Deleting and superseding one of the base functions

13.3.1 Initial Releases of Corequisite Functions

B and C are products that, together, provide a new function. These products could be packaged as base-function SYSMODs that are **unconditional corequisites**. This relationship is defined by the REQ operand on the ++VER statement for each base function, as shown in Figure 12.

In addition, a language-support dependent function is provided for each base function. Each base function is an **unconditional prerequisite** for its corresponding language-support dependent function. This relationship is defined by the FMID operand on the ++VER statement for each dependent function, as shown in Figure 12.

<table>
<tr><th>Product B</th><th>Product C</th></tr>
<tr><td><pre>++FUNCTION(WB0)
  REWORK(2003020).
++VER(Z038) REQ(WC0).
++JCLIN RELFILE(1).
++MOD(C).
++MOD(D).
++MOD(E).
++MOD(F).
++MOD(G).</pre></td><td><pre>++FUNCTION(WC0)
  REWORK(2003020).
++VER(Z038) REQ(WB0).
++JCLIN RELFILE(1).
++MOD(M).
++MOD(N).</pre></td></tr>
<tr><td><pre>++FUNCTION(XB0ENU)
  REWORK(2003020).
++VER(Z038) FMID(WB0).
++JCLIN RELFILE(1).
++PNLENU(BP).
++MSGENU(BM).</pre></td><td><pre>++FUNCTION(XC0ENU)
  REWORK(2003020).
++VER(Z038) FMID(WC0).
++JCLIN RELFILE(1).
++PNLENU(CP).
++MSGENU(CM).</pre></td></tr>
</table>

Figure 12. Initial Releases of Corequisite Functions

13.3.2 PTF Service for One of the Base Functions

Suppose you need to provide service for module C in function HB0. The fix is applicable to function HB0, so HB0 is an **unconditional prerequisite** for PTF(PB1). You define this relationship by coding a ++VER statement that specifies function HB0 as the FMID, as shown in Figure 13.

Product B
++FUNCTION(WB0)   REWORK(2003020). ++VER(Z038) REQ(WC0). ++JCLIN RELFILE(1). ++MOD(C). ++MOD(D). ++MOD(E). ++MOD(F). ++MOD(G).
++PTF(PB1). ++VER(Z038) **FMID(WB0).** ++MOD(C).

Figure 13. PTF Service for One of the Base Functions

13.3.3 Cross-Product Service between Corequisite Base Functions

Suppose you need to provide service that affects module D and module M. Module D is owned by function HB0, and module M is owned by function HC0. To fix the problem, you need two PTFs, one for each module. PB2 fixes module D, and PC1 fixes module M. These PTFs are **conditional corequisites**. (This also ensures that each PTF can still be installed if the requisite product is deleted by a new release, superseded by a new release, or both.) You define this relationship by coding the FMID and REQ operands on each PTF's ++IF statement, as shown in Figure 14.

Product B	Product C

++FUNCTION(WB0)
  REWORK(2003020).
++VER(Z038) REQ(WC0).
++JCLIN RELFILE(1).
++MOD(C).
++MOD(D).
++MOD(E).
++MOD(F).
++MOD(G).
	++FUNCTION(WC0)
	  REWORK(2003020).
	++VER(Z038) REQ(WB0).
	++JCLIN RELFILE(1).
	++MOD(M).
	++MOD(N).

++PTF(PB2). ++VER(Z038) FMID(WB0). **++IF FMID(WC0) REQ(PC1).** ++MOD(D).	++PTF(PC1). ++VER(Z038) FMID(WC0) **++IF FMID(WB0) REQ(PB2).** ++MOD(M).

Figure 14. Cross-Product Service between Corequisite Base Functions

13.3.4 Deleting and Superseding a Base Function

Suppose there are a number of improvements you want to make in product C, so you are thinking of packaging a new release. The new release could delete the initial release, supersede it, or both. (See Table 5 on page 21 if you need to review the differences.)

In this case, you have decided that function HC1 will **unconditionally delete** and **supersede** HC0. This is to ensure that requisites specified by function HB0 are satisfied by both releases of product C. You define this relationship by coding the DELETE and SUP operands on the ++VER statement for function HC1, as shown in the figure below.

You must also provide a new release of the language-support dependent function. This new release (JC1ENU) must supersede the previous release (JC0ENU), as shown in Figure 15.

Note: There is no need for HC1 or JC1ENU to delete JC0ENU. JC0ENU will automatically be deleted when HC0 is deleted. However, an explicit DELETE is recommended for documentation purposes.

Product C	
Old Release	New Release
++FUNCTION(WC0) REWORK(2003020). ++VER(Z038) REQ(WB0). ++JCLIN RELFILE(1). ++MOD(M). ++MOD(N).	++FUNCTION(WC1) REWORK(2003120). ++VER(Z038) REQ(WB0) **DELETE(WC0,XC0ENU)** **SUP(WC0).** ++JCLIN RELFILE(1). ++MOD(M). ++MOD(N).
++FUNCTION(XC0ENU) REWORK(2003020). ++VER(Z038) FMID(WC0). ++JCLIN RELFILE(1). ++PNLENU(CP). ++MSGENU(CM).	++FUNCTION(XC1ENU) REWORK(2003120). ++VER(Z038) FMID(WC1) **SUP(XC0ENU).** ++JCLIN RELFILE(1). ++PNLENU(CP). ++MSGENU(CM).

Figure 15. Deleting and Superseding a Base Function

If HC1 had only deleted HC0, instead of deleting and superseding it, any function or service that needed HC0 could not be installed without special processing. For example, users would need to have SMP/E bypass requisite checking to install the function or service. Because you know that HB0 has dependencies on HC0, you want to avoid this problem by having HC1 both delete and supersede HC0.

13.4 Example 3: Dependent Functions

In the course of developing products B and C you may decide to provide some optional enhancements that add to, but do not replace, the initial base functions. These enhancements would be packaged as dependent functions for the parent base functions, and would have their own language-support dependent functions. These are some of the SYSMODs you might have to define in the course of developing and servicing dependent functions for products B and C:

- The initial release of a dependent function. This release consists of:
  - The additive dependent function itself
  - The associated language-support dependent function
- PTF service for a dependent function
- Corequisite PTFs with an element common to the base and dependent functions
- Corequisite PTFs with no elements common to the base and dependent functions
- Repackaging a dependent function for a new release of the parent base function
- Deleting a dependent function
- Establishing the order of additional dependent functions
- Corequisite dependent functions

13.4.1 Initial Release of a Dependent Function

Suppose you have decided to provide an optional enhancement for product B. You package it as JB1, a dependent function for the parent base function HB0. Because function HB0 is the base function, it is an **unconditional prerequisite** for function JB1. You define this relationship by coding the FMID operand on the dependent function's ++VER statement, as shown in Figure 16.

Product B	
Base Function and Language Support	Dependent Function and Language Support

++FUNCTION(WB0)
  REWORK(2003020).
++VER(Z038) REQ(WC0).
++JCLIN RELFILE(1).
++MOD(C).
++MOD(D).
++MOD(E).
++MOD(F).
++MOD(G).
	++FUNCTION(XB1)
	  REWORK(2003070).
	++VER(Z038) **FMID(WB0).**
	++JCLIN RELFILE(1).
	++MOD(F).
	++MOD(H).
	++MOD(J).

++FUNCTION(XB0ENU)
  REWORK(2003020).
++VER(Z038) FMID(WB0).
++JCLIN RELFILE(1).
++PNLENU(BP).
++MSGENU(BM).
++CLISTENU(BC).
	++FUNCTION(XB1ENU)
	  REWORK(2003070).
	++VER(Z038) FMID(WB0)
	  PRE(XB1,XB0ENU)
	  VERSION(XB0ENU).
	++JCLIN RELFILE(1).
	++PNLENU(BP).
	++MSGENU(BM).

Figure 16. Initial Release of a Dependent Function

13.4.2 PTF Service for a Dependent Function

Suppose you need to provide service for module H in function JB1. The fix is applicable to function JB1, so JB1 is an **unconditional prerequisite** for PTF(PB3). You define this relationship by coding a ++VER statement that specifies function JB1 as the FMID, as shown in Figure 17.

```
Product B
------------------------------
++FUNCTION(XB1)
  REWORK(2003070).
++VER(Z038) FMID(WB0).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(H).
++MOD(J).
------------------------------
++PTF(PB3).
++VER(Z038) FMID(XB1).
++MOD(H).
```

Figure 17. PTF Service for a Dependent Function

13.4.3 Corequisite PTFs with an Element Common to the Base and Dependent Functions

Suppose you need to provide service that affects module F and module G. Module G is owned by base function HB0, and module F exists in HB0 and in its dependent function JB1. To fix the problem, you need two PTFs, one for each function. Because the dependent function may be installed with the base function, the PTF for the dependent function is a conditional requisite in the PTF for the base function. However, because the base function must be installed if the dependent function is installed, the PTF for the base function is an unconditional requisite in the PTF for the dependent function. You define these relationships by coding the FMID and REQ operands on the ++IF statement for the base function PTF, and by coding the REQ operand on the ++VER statement for the dependent function PTF, as shown in Figure 18 on page 146.

Product B	
Base Function and Language Support	Dependent Function and Language Support

++FUNCTION(WB0)
  REWORK(2003020).
++VER(Z038) REQ(WC0).
++JCLIN RELFILE(1).
++MOD(C).
++MOD(D).
++MOD(E).
++MOD(F).
++MOD(G).
	++FUNCTION(XB1)
	  REWORK(2003070).
	++VER(Z038) FMID(WB0).
	++JCLIN RELFILE(1).
	++MOD(F).
	++MOD(H).
	++MOD(J).

++PTF(PB4). ++VER(Z038) FMID(WB0). **++IF FMID(XB1) REQ(PB5).** ++MOD(F). ++MOD(G).	++PTF(PB5). ++VER(Z038) FMID(XB1) **REQ(PB4).** ++MOD(F).

Figure 18. Corequisite PTFs with an Element Common to the Base and Dependent Functions

Suppose functions HB0 and JB1 were released and a new dependent function JB2 is being packaged that deletes JB1. Figure 19 on page 147 shows how function JB2 and its related language-support dependent function (JB2ENU) are then packaged.

Product B	
Base Function and Language Support	Dependent Function and Language Support
++FUNCTION(WB0) REWORK(2003020). ++VER(Z038) REQ(WC0). ++JCLIN RELFILE(1). ++MOD(C). ++MOD(D). ++MOD(E). ++MOD(F). ++MOD(G).	++FUNCTION(XB1) REWORK(2003070). ++VER(Z038) FMID(WB0). ++JCLIN RELFILE(1). ++MOD(F). ++MOD(H). ++MOD(J).
++FUNCTION(XB0ENU) REWORK(2003020). ++VER(Z038) FMID(WB0). ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM). ++CLISTENU(BC).	++FUNCTION(XB1ENU) REWORK(2003070). ++VER(Z038) FMID(WB0) PRE(XB1,XB0ENU) VERSION(XB0ENU). ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM).
++PTF(PB4). ++VER(Z038) FMID(WB0). ++IF FMID(XB1) REQ(PB5). ++MOD(F). ++MOD(G).	++PTF(PB5). ++VER(Z038) FMID(XB1) REQ(PB4). ++MOD(F).
	++FUNCTION(XB2) REWORK(2003110). ++VER(Z038) FMID(WB0) **DELETE(XB1,XB1ENU)** **SUP(XB1,PB4,PB5).** ++JCLIN RELFILE(1). ++MOD(F). ++MOD(G). ++MOD(H).
	++FUNCTION(XB2ENU) REWORK(2003110). ++VER(Z038) FMID(WB0) **PRE(XB2,XB0ENU)** **VERSION(XB0ENU)** **SUP(XB1ENU).** ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM).

Figure 19. New Releases of the Base and Dependent Functions

Notes:

1. Function JB2 deletes and supersedes the previous release of the dependent function (JB1). It also deletes the language-support dependent function (JB1ENU) associated with that previous release.
2. Function JB2 does not have to refer to PTF(PB5) because function JB1 is deleted. However, JB2 does have to supersede PTF(PB4) to make sure that PB4 is not reprocessed by SMP/E.
3. Function JB2ENU supersedes the previous release of the language-support dependent function (JB1ENU).
4. If PTF(PB4) and PTF(PB5) affect two different elements, and the corequisite relationship is still required, the logic is the same.

13.4.4 Corequisite PTFs with All Elements Common to Base and Dependent Functions

Suppose you need to provide service that affects module F, which is present in both base function HB0 and dependent function JB1. To fix the problem, you need two PTFs, one for each function. Because the dependent function can be installed with the base function, the PTF for the dependent function is a **conditional corequisite** of the PTF for the base function.

If the user has the dependent function installed, the PTF for the base function really is not necessary, because the PTF for the dependent function provides a higher level of the element. However, it is important to prevent the user from accidentally installing the PTF for the base function and later downleveling the dependent function's version of the element. Therefore, the PTF for the base function is an **unconditional corequisite** of the PTF for the dependent function.

You define these relationships by coding the FMID and REQ operands on the ++IF statement for the base function PTF, and by coding either the REQ operand or the SUP operand on the ++VER statement for the dependent function PTF, as shown in Figure 20 on page 149.

<table>
<tr><th colspan="2">Product B</th></tr>
<tr><td>Base Function and
Language Support</td><td>Dependent Function and
Language Support</td></tr>
<tr><td><pre>++FUNCTION(WB0)
  REWORK(2003020).
++VER(Z038) REQ(WC0).
++JCLIN RELFILE(1).
++MOD(C).
++MOD(D).
++MOD(E).
++MOD(F).
++MOD(G).</pre></td><td><pre>++FUNCTION(XB1)
  REWORK(2003070).
++VER(Z038) FMID(WB0).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(H).
++MOD(J).</pre></td></tr>
<tr><td><pre>++FUNCTION(XB0ENU)
  REWORK(2003020).
++VER(Z038) FMID(WB0).
++JCLIN RELFILE(1).
++PNLENU(BP).
++MSGENU(BM).</pre></td><td><pre>++FUNCTION(XB1ENU)
  REWORK(2003070).
++VER(Z038) FMID(WB0)
  PRE(XB1,XB0ENU)
  VERSION(XB0ENU).
++JCLIN RELFILE(1).
++PNLENU(BP).
++MSGENU(BM).</pre></td></tr>
<tr><td><pre>++PTF(PB6).
++VER(Z038) FMID(WB0).
++IF FMID(XB1) REQ(PB7).
++MOD(F).</pre></td><td><pre>++PTF(PB7).
++VER(Z038) FMID(XB1)
            REQ(PB6).
++MOD(F).</pre></td></tr>
</table>

Figure 20. Corequisite PTFs with All Elements Common to Base and Dependent Functions

Suppose functions HB0 and JB1 were released and a new dependent function JB2 is being packaged that deletes JB1, as shown in Figure 21 on page 150.

Product B	
Base Function and Language Support	Dependent Function and Language Support
++FUNCTION(WB0) REWORK(2003020). ++VER(Z038) REQ(WC0). ++JCLIN RELFILE(1). ++MOD(C). ++MOD(D). ++MOD(E). ++MOD(F). ++MOD(G).	++FUNCTION(XB1) REWORK(2003070). ++VER(Z038) FMID(WB0). ++JCLIN RELFILE(1). ++MOD(F). ++MOD(H). ++MOD(J).
++FUNCTION(XB0ENU) REWORK(2003020). ++VER(Z038) FMID(WB0). ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM).	++FUNCTION(XB1ENU) REWORK(2003070). ++VER(Z038) FMID(WB0) PRE(XB1,XB0ENU) VERSION(XB0ENU). ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM).
++PTF(PB6). ++VER(Z038) FMID(WB0). ++IF FMID(XB1) REQ(PB7). ++MOD(F).	++PTF(PB7). ++VER(Z038) FMID(XB1) REQ(PB6). ++MOD(F).
	++FUNCTION(XB2) REWORK(2003110). ++VER(Z038) FMID(WB0) **DELETE(XB1,XB1ENU)** **SUP(XB1)** **SUP(PB6).** ++JCLIN RELFILE(1). ++MOD(F). ++MOD(H). ++MOD(J). ++MOD(K).
	++FUNCTION(XB2ENU) REWORK(2003110). ++VER(Z038) FMID(WB0) PRE(XB2,XB0ENU) VERSION(XB0ENU) SUP(XB1ENU). ++JCLIN RELFILE(1). ++PNLENU(BP). ++MSGENU(BM).

Figure 21. New Releases of the Base and Dependent Functions

Note: Function JB2 does not have to refer to PTF(PB7), because function JB1 is deleted. However, JB2 does have to supersede PTF(PB6) to prevent PB6 from being reprocessed.

13.4.5 Deleting a Dependent Function Without Superseding It

Suppose function HB7 is a new release of dependent function HB6. HB7 changes the external interface of the dependent function so it is no longer compatible with prior releases. HB7 would, therefore, **unconditionally delete** HB6 but must not supersede HB6. You define this relationship by coding the DELETE operand on the ++VER statement for function HB7, as shown in Figure 22.

Product B	
Old Release	New Release

++FUNCTION(WB6)
  REWORK(2003210).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(H).
++MOD(J).
++MOD(K).
	++FUNCTION(WB7)
	  REWORK(2003240).
	++VER(Z038)
	  DELETE(WB6).
	++JCLIN RELFILE(1).
	++MOD(F).
	++MOD(H).
	++MOD(J).
	++MOD(K).

Figure 22. Deleting a Dependent Function

13.4.6 Establishing the Order of Additional Dependent Functions

Suppose you have added an optional enhancement for product B. It is packaged as function JB8, a dependent function. JB8 does not delete or supersede any other dependent functions. However, because it has a requirement for modules in JB7, another dependent function, you must establish which of the dependent functions depends on the other. For example, if function JB8 is functionally higher than function JB7, function JB7 is an **unconditional prerequisite** for function JB8. You define this relationship by coding the PRE operand on function JB8's ++VER statement, as shown in Figure 23.

Product B	
Lower Level	Higher Level

++FUNCTION(XB7)
  REWORK(2003240).
++VER(Z038) FMID(WB5).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(H).
++MOD(J).
++MOD(K).
	++FUNCTION(XB8)
	  REWORK(2003260).
	++VER(Z038) FMID(WB5)
	            PRE(XB7)
	            VERSION(XB7).
	++MOD(F).

Figure 23. Establishing the Order of Additional Dependent Functions

Note: The VERSION operand is required in functions JB8 and JB8ENU to change ownership of the common elements MOD(F) and PNLENU(BP). See 7.2, "++VER Statement" on page 53 for more information.

13.4.7 Conditional Corequisite Dependent Functions

Suppose you have developed a new user function that involves elements from dependent functions JB9 and JC2. JB9 is a dependent function for base function HB5, and JC2 is a dependent function for base function HC1. The code works properly only if the two dependent functions are installed together. These dependent functions are **conditional corequisites**. (This also ensures that either dependent function can still be installed if the other dependent function's parent base function is deleted by a new release, superseded by a new release, or both.) You define this relationship by coding the FMID and REQ operands on each dependent function's ++IF statements, as shown in Figure 24.

Product B	Product C

++FUNCTION(XB9)
  REWORK(2003300).
++VER(Z038) FMID(WB5).
++IF FMID(WC1) REQ(XC2).
++MOD(L).
	++FUNCTION(XC2)
	  REWORK(2003300).
	++VER(Z038) FMID(WC1).
	++IF FMID(WB5) REQ(XB9).
	++MOD(N).

Figure 24. Corequisite Dependent Functions

13.5 Example 4: Base Functions with Prerequisites

Functions may depend on other functions as prerequisites, or they may depend on service provided for another function. Products C, D, and E are examples of these. Product D depends on product C; product E depends on service for product D. These are some of the relationships you may define in the course of developing and servicing these products:

- The initial release of a base function with a functional prerequisite. This release consists of:
  - The base function itself
  - The associated language-support dependent function
- Dependency on an SPE or service for another base function
- Cross-product service for a base function with a prerequisite

13.5.1 Initial Release of a Base Function with a Functional Prerequisite

Suppose your base function HC0 for product C provides the minimum level of support for base function HD0, the first release of product D. Function HC0 is, therefore, an **unconditional prerequisite** for function HD0. The owner of function HD0 defines this relationship by specifying the REQ operand on the ++VER statement for HD0, as shown in Figure 25 on page 153.

Product C	Product D

++FUNCTION(WC0)
  REWORK(2003020).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(M).
++MOD(N).
	++FUNCTION(WD0)
	  REWORK(2003060).
	++VER(Z038) REQ(WC0).
	++JCLIN RELFILE(1).
	++MOD(P).
	++MOD(Q).

++FUNCTION(XC0ENU)
  REWORK(2003020).
++VER(Z038) FMID(WC0).
++JCLIN RELFILE(1).
++PNLENU(CP).
++MSGENU(CM).
	++FUNCTION(XD0ENU)
	  REWORK(2003060).
	++VER(Z038) FMID(WD0).
	++JCLIN RELFILE(1).
	++PNLENU(DP).
	++MSGENU(DM).

Figure 25. Initial Release of a Base Function with a Functional Prerequisite

Suppose you come out with a new release of the base function, HC1. If HC0 is both deleted and superseded by HC1, as shown in Figure 26, HD0 does not need to be repackaged to work with both releases of product C.

Product C		
Old Release	New Release	Product D

++FUNCTION(WC0)
  REWORK(2003020).
++VER(Z038) REQ(WB0).
++JCLIN RELFILE(1).
++MOD(M).
++MOD(N).
	++FUNCTION(WC1)
	  REWORK(2003120).
	++VER(Z038) REQ(WB0)
	  DELETE(WC0)
	  SUP(WC0).
	++JCLIN RELFILE(1).
	++MOD(M).
	++MOD(N).
		++FUNCTION(WD0)
		  REWORK(2003060).
		++VER(Z038) REQ(WC0).
		++JCLIN RELFILE(1).
		++MOD(P).
		++MOD(Q).

Figure 26. New Release of a Base Function with a Functional Prerequisite

Note: The owner of product D must consider that future releases of product C may not be compatible with HD0. For example, they may not provide the required support the same way HC0 or HC1 did. This may not be a problem if they have a part in the development or packaging of your product C. However, if this is not the case, they may have to change or service product D to keep up with the support provided by your new releases of C.

13.5.2 Dependency on an SPE or Service for Another Base Function

Suppose you have provided a small programming enhancement (SPE) or service for function HD0. This service is packaged as PTF(PD1). You have also developed a new product, which will be packaged as base function HE0. When function HE0 interacts with function HD0, it requires PD1. PD1 is, therefore, a **conditional prerequisite** for function HE0. You define this relationship by coding an ++IF statement for PD1 in function HE0, as shown in Figure 27 on page 154.

Product D	Product E
++FUNCTION(WD0) REWORK(2003060). ++VER(Z038). ++JCLIN RELFILE(1). ++MOD(P). ++MOD(Q).	++FUNCTION(WE0) REWORK(2003090). ++VER(Z038). **++IF FMID(WD0) REQ(PD1).** ++JCLIN RELFILE(1). ++MOD(R). ++MOD(S).
++PTF(PD1). ++VER(Z038) FMID(WD0). ++MOD(P).	

Figure 27. Dependency on an SPE or Service for Another Base Function

Note: Any replacement for PTF(PD1) must supersede PD1 to ensure that this requisite for function HE0 is still satisfied.

13.5.3 Cross-Product Service for a Base Function with a Prerequisite

Suppose you need to provide service that affects module Q and module R. Module Q is owned by function HD0, and module R is owned by function HE0. To fix the problem, you need two PTFs, one for each module. PD2 fixes module Q, and PE1 fixes module R. Because the functions may be installed with or without each other, the PTFs for those functions are **conditional corequisites**. You define this relationship by coding an ++IF statement in each PTF, as shown in Figure 28.

Product D	Product E
++FUNCTION(WD0) REWORK(2003060). ++VER(Z038). ++JCLIN RELFILE(1). ++MOD(P). ++MOD(Q).	++FUNCTION(WE0) REWORK(2003090). ++VER(Z038). ++IF FMID(WD0) REQ(PD1). ++JCLIN RELFILE(1). ++MOD(R). ++MOD(S).
++FUNCTION(XD0ENU) REWORK(2003060). ++VER(Z038) FMID(WD0). ++JCLIN RELFILE(1). ++PNLENU(DP). ++MSGENU(DM).	++FUNCTION(XE0ENU) REWORK(2003090). ++VER(Z038) FMID(WE0). ++JCLIN RELFILE(1). ++PNLENU(EP). ++MSGENU(EM).
++PTF(PD1). ++VER(Z038) FMID(WD0). ++MOD(P).	
++PTF(PD2). ++VER(Z038) FMID(WD0) PRE(PD1). **++IF FMID(WE0) REQ(PE1).** ++MOD(Q).	++PTF(PE1). ++VER(Z038) FMID(WE0). **++IF FMID(WD0) REQ(PD2).** ++MOD(R).

Figure 28. Cross-Product Service for a Base Function with a Prerequisite

13.6 Example 5: Mutually Exclusive Dependent Functions

Suppose function JE1 and function JE2 are dependent functions for the same base function, HE0. Function JE1 tailors the base to one specific environment, and function JE2 tailors it to another specific environment. Because these SYSMODs provide mutually exclusive functions, they are **unconditional negative prerequisites** of each other. You define this relationship by coding the NPRE operand on each function's ++VER statement, as shown in Figure 29.

```
                         Product E

Base Function and            Dependent Functions and
Language Support             Language Support

++FUNCTION(WE0)              ++FUNCTION(XE1)
  REWORK(2003090).             REWORK(2003130).
++VER(Z038).                 ++VER(Z038) FMID(WE0)
++IF FMID(WD0) REQ(PD1).                 NPRE(XE2).
++JCLIN RELFILE(1).          ++MOD(R).
++MOD(R).                    ++MOD(S).
++MOD(S).
                             ---------- and ----------
++FUNCTION(XE0ENU)           ++FUNCTION(XE1ENU)
  REWORK(2003090).             REWORK(2003130).
++VER(Z038) FMID(WE0).       ++VER(Z038) FMID(WE0)
                                 NPRE(XE2ENU)
                                 PRE(XE1,XE0ENU)
                                 VERSION(XE0ENU).
++JCLIN RELFILE(1).          ++JCLIN RELFILE(1).
++PNLENU(EP).                ++PNLENU(EP).
++MSGENU(EM).                ++MSGENU(EM).
                             ----------(or)-----------
                             ++FUNCTION(XE2)
                               REWORK(2003130).
                             ++VER(Z038) FMID(WE0)
                                         NPRE(XE1).
                             ++MOD(R).
                             ++MOD(S).
                             ---------- and ----------
                             ++FUNCTION(XE2ENU)
                               REWORK(2003130).
                             ++VER(Z038) FMID(WE0)
                               NPRE(XE1ENU)
                               PRE(XE2,XE0ENU)
                               VERSION(XE0ENU).
                             ++JCLIN RELFILE(1).
                             ++PNLENU(EP).
                             ++MSGENU(EM).
```

Figure 29. Mutually Exclusive Dependent Functions

13.7 Example 6: Functions Supporting More Than One Language

As shown in the previous sections, any language support you provide for a function must be packaged in a language-support dependent function.

In the course of developing a product, you may decide to provide support for additional languages. For each additional language, the language-sensitive elements must also be packaged in a separate language-support dependent function.

These are some of the situations with relationships you might have to define in the course of developing and servicing dependent functions to support more than one language:

- Supporting two languages for a base function
- Providing PTF service for language-sensitive elements
- Supporting two languages for a base function and its related dependent function
- Providing PTF service for common language-sensitive elements
- Providing PTF service for language-sensitive elements unique to the dependent function

13.7.1 A Base Function Supporting Two Languages

Suppose you have a product that must provide information (such as messages and dialog elements) in both U.S. English and French. The language-sensitive elements for each language must be packaged in a separate dependent function for each language. The remaining elements are packaged in the base function (HX0). Because function HX0 is the base function, it is an **unconditional prerequisite** for the language-support dependent functions (JX0ENU and JX0FRA). You define this relationship by coding the FMID operand on the ++VER statements in each dependent function, as shown in Figure 30 on page 157.

Product X	
Base Function	Language Support

++FUNCTION(WX0)
  REWORK(2003140).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(U).
++MOD(V).
	++FUNCTION(XX0ENU)
	  REWORK(2003140).
	++VER(Z038) **FMID(WX0).**
	++JCLIN RELFILE(1).
	++PNLENU(XP)
	  DISTLIB(AXXXPENU)
	  SYSLIB(SXXXPENU).

	++FUNCTION(XX0FRA)
	  REWORK(2003140).
	++VER(Z038) **FMID(WX0).**
	++JCLIN RELFILE(1).
	++PNLFRA(XP)
	  DISTLIB(AXXXPFRA)
	  SYSLIB(SXXXPFRA).

Note: In this example, DISTLIB and SYSLIB values were specified for panel XP to emphasize that language-sensitive elements must be packaged in unique distribution and target libraries. JCLIN was not necessary.

Figure 30. A Base Function Supporting Two Languages

13.7.2 PTF Service for Language-Sensitive Elements

Suppose you need to correct a mistake that exists in panel XP for both dependent functions (JX0ENU and JX0FRA). You need to provide a separate PTF for each dependent function. Because the dependent functions are independent of each other, no relationship needs to be defined between these PTFs. Because the change only affects language-sensitive elements, no PTF is required for the base function. Each dependent function is an **unconditional prerequisite** for its associated PTF. You define this relationship by coding the appropriate FMID on the ++VER statement for each PTF, as shown in Figure 31.

Product X	
English Support	French Support

++FUNCTION(XX0ENU)
  REWORK(2003140).
++VER(Z038) FMID(WX0).
++JCLIN RELFILE(1).
++PNLENU(XP).
	++FUNCTION(XX0FRA)
	  REWORK(2003140).
	++VER(Z038) FMID(WX0).
	++JCLIN RELFILE(1).
	++PNLFRA(XP).

++PTF(PX1). **++VER(Z038) FMID(XX0ENU).** ++JCLIN RELFILE(1). ++PNLENU(XP).	++PTF(PX2). **++VER(Z038) FMID(XX0FRA).** ++JCLIN RELFILE(1). ++PNLFRA(XP).

Figure 31. PTF Service for Language-Sensitive Elements

13.7.3 Supporting Two Languages for a Base Function and Its Related Dependent Function

Suppose you have a product consisting of a base function plus a dependent function for an optional enhancement. You want to provide support for messages and dialogs in both English and French for the base function and the dependent function.

The language-sensitive elements for each language must be packaged in a separate dependent function for each language. As shown in Figure 32 on page 159, you need two dependent functions to support the language-sensitive elements for the base function, and two more to support the optional dependent function.

- The base function (HY0) is an **unconditional prerequisite** for its language support functions JY0ENU and JY0FRA. It is also an **unconditional prerequisite** for JY1, its dependent function for the optional enhancement.

  You define these relationships by coding the FMID operand on the ++VER statements in JY0ENU, JY0FRA, and JY1.

- JY1 is an **unconditional prerequisite** for its language support functions JY1ENU and JY1FRA.

  You define this relationship by coding the PRE operand on the ++VER statements in JY1ENU and JY1FRA.

- Because language-support dependent functions JY0ENU and JY0FRA are applicable to base function HY0, they are unconditional prerequisites for language-support dependent functions JY1ENU and JY1FRA, which are applicable to dependent function JY1.

  You define this relationship by coding the PRE operand on the ++VER statements in JY1ENU and JY1FRA.

<table>
<tr><th colspan="2">Product Y</th></tr>
<tr><td>Base Function and
Language Support</td><td>Dependent Function and
Language Support</td></tr>
<tr><td><pre>++FUNCTION(WY0)
  REWORK(2003250).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(X).
++MOD(Y).</pre></td><td><pre>++FUNCTION(XY1)
  REWORK(2003250).
++VER(Z038) FMID(WY0).
++MOD(X).
++MOD(Y).</pre></td></tr>
<tr><td><pre>++FUNCTION(XY0ENU)
  REWORK(2003250).
++VER(Z038) FMID(WY0).
++JCLIN RELFILE(1).
++PNLENU(Y0P).</pre></td><td><pre>++FUNCTION(XY1ENU)
  REWORK(2003250).
++VER(Z038) FMID(WY0)
  PRE(XY1,XY0ENU)
  VERSION(XY0ENU).
++JCLIN RELFILE(1).
++PNLENU(Y0P).
++PNLENU(Y1P).</pre></td></tr>
<tr><td><pre>++FUNCTION(XY0FRA)
  REWORK(2003250).
++VER(Z038) FMID(WY0).
++JCLIN RELFILE(1).
++PNLFRA(Y0P).</pre></td><td><pre>++FUNCTION(XY1FRA)
  REWORK(2003250).
++VER(Z038) FMID(WY0)
  PRE(XY1,XY0FRA)
  VERSION(XY0FRA).
++JCLIN RELFILE(1).
++PNLFRA(Y0P).
++PNLFRA(Y1P).</pre></td></tr>
</table>

Figure 32. Supporting Two Languages for a Base Function and Its Related Dependent Function

Note: The VERSION operand is required to change ownership of the elements. See 7.2, "++VER Statement" on page 53 and Chapter 6, "Elements and Load Modules" on page 37 for more information.

13.7.4 PTF Service for Common Language-Sensitive Elements

Suppose you need to provide service that affects panel Y0P. There are four versions of this panel: an English and a French version for the base function, and an English and a French version for the dependent function. Each of these versions is owned by a different dependent function for language-sensitive elements. Therefore, to fix the problem, you need four PTFs, one for each of the dependent functions for language support, as shown in Figure 33 on page 160. You must provide ++IF REQ statements to define the relationship between related PTFs for the same language. However, you do not need to define any relationship between PTFs for different languages.

Product Y	
English Support for the Base Function (WY0)	English Support for the Dependent Function (XY1)

++FUNCTION(XY0ENU)
  REWORK(2003250).
++VER(Z038) FMID(WY0).
++PNLENU(Y0P).
	++FUNCTION(XY1ENU)
	  REWORK(2003250).
	++VER(Z038) FMID(WY0)
	  PRE(XY1,XY0ENU)
	  VERSION(XY0ENU).
	++JCLIN RELFILE(1).
	++PNLENU(Y0P).
	++PNLENU(Y1P).

++PTF(PY2). ++VER(Z038) FMID(XY0ENU). **++IF FMID(XY1ENU) REQ(PY4).** ++PNLENU(Y0P).	++PTF(PY4). ++VER(Z038) FMID(XY1ENU). ++PNLENU(Y0P).

Product Y	
French Support for the Base Function (WY0)	French Support for the Dependent Function (XY1)

++FUNCTION(XY0FRA)
  REWORK(2003250).
++VER(Z038) FMID(WY0).
++PNLFRA(Y0P).
	++FUNCTION(XY1FRA)
	  REWORK(2003250).
	++VER(Z038) FMID(WY0)
	  PRE(XY1,XY0FRA)
	  VERSION(XY0FRA).
	++JCLIN RELFILE(1).
	++PNLFRA(Y0P).
	++PNLFRA(Y1P).

++PTF(PY3). ++VER(Z038) FMID(XY0FRA). **++IF FMID(XY1FRA) REQ(PY5).** ++PNLFRA(Y0P).	++PTF(PY5). ++VER(Z038) FMID(XY1FRA). ++PNLFRA(Y0P).

Figure 33. PTF Service for Common Language-Sensitive Elements

13.8 Changing the Contents of Products

After the elements that make up a product have been defined, changes in the contents of the product may be required. For example, an element may need to be added, deleted, combined with another element, or moved to another product. The following sections provide information to help you decide how to make these changes.

13.8.1 Adding Elements

You can add elements to a product using a new release of a base or dependent function, or using a PTF. When a new base or dependent function release adds elements, previous releases of the function are not affected. The new elements are serviced as long as the functions that own them are current. in Figure 34, dependent function JK1 adds module C to product K.

Product K	
Base Function	Dependent Function

++FUNCTION(WK0)
  REWORK(2003170).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).
++MOD(T).
	++FUNCTION(XK1)
	  REWORK(2003200).
	++VER(Z038) FMID(WK0).
	++JCLIN RELFILE(1).
	++MOD(J).
	++MOD(K).
	++MOD(C).

Figure 34. Adding Elements

Notes:

1. The ownership of MOD(J) and MOD(K) is transferred to function JK1. (See 7.2.8, “Defining Ownership (VERSION)” on page 62 for versioning rules.)
2. The ++JCLIN statement and JCLIN data are required to define the revised load module structure.

When a PTF adds elements, it specifies the function that is to own the new elements. If the load module structure is changed, the PTF may also include new JCLIN.

13.8.2 Combining Elements

You can combine elements in a product using a new release of a base or dependent function, or using a PTF. For example, instead of using two modules to provide a given user function, you may combine all the function into one module, and delete the other one.

A new base or dependent function may combine and delete elements that existed in a previous release. However, service must continue to be provided for both versions of the elements during the service currency of the previous release of the product.

In Figure 35 on page 162, dependent function JK2 combines modules A and B, deleting module B from prerequisite dependent function JK1.

<table>
<tr><th colspan="2">Product K</th></tr>
<tr><th>Base Function</th><th>Dependent Functions</th></tr>
<tr><td>++FUNCTION(WK0)
  REWORK(2003170).
++VER(Z038).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).
++MOD(T).</td><td>++FUNCTION(XK1)
  REWORK(2003200).
++VER(Z038) FMID(WK0).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).
++MOD(C).</td></tr>
<tr><td></td><td>++FUNCTION(XK2)
  REWORK(2003230).
++VER(Z038) FMID(WK0)
          PRE(XK1).
++JCLIN RELFILE(1).
++MOD(J) VERSION(XK1).
++MOD(K) DELETE.</td></tr>
</table>

Figure 35. Combining Elements

13.8.3 Migrating Elements by Updating Both Functions

This method is straightforward and is the recommended way of migrating elements from one function to another. The element is deleted from one function and added to another. The new releases of the functions are issued simultaneously and must be installed concurrently. In Figure 36, the new base function release HL1 no longer contains module H, which is now included in the new base function release M1.

<table>
<tr><th>Product L</th><th>Product M</th></tr>
<tr><td>++FUNCTION(WL0)
  REWORK(2003180).
++VER(Z038) REQ(WM0).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(G).
++MOD(H).</td><td>++FUNCTION(WM0)
  REWORK(2003180).
++VER(Z038) REQ(WL0).
++JCLIN RELFILE(1).
++MOD(J).
++MOD(K).</td></tr>
<tr><td>++FUNCTION(WL1)
  REWORK(2003280).
++VER(Z038) DELETE(WL0)
          REQ(WM1).
++JCLIN RELFILE(1).
++MOD(F).
++MOD(G).</td><td>++FUNCTION(WM1)
  REWORK(2003280).
++VER(Z038) DELETE(WM0)
          REQ(WL1).
++JCLIN RELFILE(1).
++MOD(H).
++MOD(J).
++MOD(K).</td></tr>
</table>

Figure 36. Migrating Elements by Updating Both Functions

13.8.4 Migrating Elements by Using a PTF

A PTF can provide new versions of elements for a function, as well as specify which elements are now owned by that function, and which functions had previously owned those elements. All subsequent releases of the functions affected by the migration must reflect the changes made by the PTF.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

BookManager
CBPDO
IBM
IBMLink
MVS/DFP
MVS/ESA
MVS/SP
MVS/XA
ServerPac
SystemPac
System/370
OS/390
z/OS

© Copyright IBM Corp. 1986, 2003

Glossary

This glossary defines terms and abbreviations used in this publication.

Sequence of Entries: For clarity and consistency of style, this glossary arranges the entries alphabetically on a letter-by-letter basis. In other words, only the letters of the alphabet are used to determine sequence; special characters and spaces between words are ignored.

Organization of Entries: Each entry consists of a single-word or multiple-word term or the abbreviation or acronym for a term, followed by a commentary. A commentary includes one or more items (definitions or references) and is organized as follows:

1. An item number, if the commentary contains two or more items.
2. A usage label, indicating the area of application of the term, for example, “In programming,” or “In SMP/E.” Absence of a usage label implies that the term is generally applicable to SMP/E, to IBM, or to data processing.
3. A descriptive phrase, stating the basic meaning of the term. The descriptive phrase is assumed to be preceded by “the term is defined as ...” The part of speech being defined is indicated by the opening words of the descriptive phrase: “To ...” indicates a verb, and “Pertaining to ...” indicates a modifier. Any other wording indicates a noun or noun phrase.
4. Annotative sentences, providing additional or explanatory information.
5. References, directing the reader to other entries or items in the dictionary.

References: The following cross-references are used in this glossary:

Contrast with. This refers to a term that has an opposed or substantively different meaning.

Synonym for. This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the glossary.

Synonymous with. This is a backward reference from a defined term to all other terms that have the same meaning.

See. This refers you to multiple-word terms that have the same last word.

See also. This refers the reader to related terms that have a related, but not synonymous, meaning.

Deprecated term for or **Deprecated abbreviation for.** This indicates that the term or abbreviation should not be used. It refers to a preferred term, which is defined in its proper place in the glossary.

Selection of Terms: A term is a word or group of words to be defined. In this glossary, the singular form of the noun and the infinitive form of the verb are the terms most often selected to be defined. If the term may be abbreviated, the abbreviation is given in parentheses immediately following the term. The abbreviation is also defined in its proper place in the glossary.

A

ACCEPT. In SMP/E, the process initiated by the ACCEPT command that places SYSMODs into the distribution libraries or permanent user libraries.

APAR. Authorized program analysis report.

APAR fix. A temporary correction of a defect in an IBM software product. An APAR fix is usually replaced at a later date by a permanent correction (PTF). In SMP/E, APAR fixes are identified by the ++APAR statement. APARs incorporated into PTFs and function SYSMODs are specified in the ++VER SUP operand.

APPLY. In SMP/E, the process, initiated by the APPLY command, that places SYSMODs into the target libraries.

authorized program analysis report (APAR). A report of a problem caused by a suspected defect in a current unaltered release of a program. The correction is called an APAR fix.

B

base function. A SYSMOD that defines elements of the base system or other products that were not previously present in the target libraries. Base functions are identified to SMP/E by the ++FUNCTION statement.

base system. The set of product functions defined as the minimum set required to form an operational software system (for example, the minimum set of product functions to form an MVS/ESA operating system).

basic materials. These are the RELFILE tape, program directory, and publications necessary to install and operate a product function.

binder. A program that processes the output of language translators and compilers into an executable

© Copyright IBM Corp. 1986, 2003

program (load module or program object). Part of DFSMS/z/OS, it replaces the linkage editor and batch loader.

BYPASS. In SMP/E, an ACCEPT command operand that bypasses certain conditions to allow SMP/E processing of a SYSMOD to continue, regardless of the existence of that condition. For example, BYPASS (APPLYCHECK) indicates that all SYSMODs found in the PTS and not yet accepted are eligible for ACCEPT processing even if they have not been applied.

C

coexisting functions. Functions that can reside on the same system and be described by the same target zone.

common code. Code that is source line identical for two or more environments.

common element. An element that is part of two different functions. It has the same name and type in each function. See also *element intersection*.

COMP ID. Component identifier.

component. Components are subdivisions of product functions (that is, a grouping of elements within a product). Components are a means of splitting a product function for workload planning, tracking, and servicing purposes.

component code. Used as the first three characters of external (element and library) names.

component name. See *component identifier*.

conditionally coexisting functions. Functions that coexist but do not have to be in the same zone.

consolidated software inventory (CSI). The primary SMP/E data set, which is divided into multiple partitions called zones. The three types of zones are the global zone, target zone, and distribution zone. A zone may be associated with one or more SRELs. Each zone contains information necessary for defining a system or subsystem and processing data for installing product function and service SYSMODs onto that system or subsystem.

corrective service. Corrective service is provided to a customer who is experiencing a high-impact problem. The corrective fix is an expedited fix, thus, it may not go through the quality procedures required for preventive service. The preferred format for a corrective service fix is the PTF. However, other appropriate formats, bypasses, or circumventions are also an appropriate response.

cross-system product (CSP). A product that is supported on more than one operating system and has specific dependencies on any product functions or their service levels in one or more of the operating systems on which the product is supported.

CSI. Consolidated software inventory.

CSI distribution zone. See *distribution zone*.

CSI target zone. See *target zone*.

CSP. Cross-system product

CUM. Cumulative service tape.

cumulative service tape (CUM). The tape sent with a new product function order, containing all current non-integrated PTFs for that product function.

currency. The duration of time for which a product function release receives service support.

customization. Jobs or procedures required after "installation" before a product's function can be used, or before a product's service is effective.

D

data element. An element that is not a macro, module, or source code; for example, a dialog panel or sample code.

dependent function. A function that introduces new elements or redefines elements of the base-level system or other products. A dependent function cannot exist without a base function. Dependent functions are identified to SMP/E by the ++FUNCTION statement.

development process. Develops product function in response to marketing, user, and service requirements.

DFSMS environment. An environment that helps automate and centralize the management of storage. This is achieved through a combination of hardware, software, and policies. In the DFSMS environment for z/OS, this function is provided by MVS/ESA SP and DFSMS/MVS, DFSORT, and RACF.

distribution class. The terms of distribution for a product function, which include:

- Non-Restricted Materials - The product function is available with source materials.
- Restricted Materials - The product function is available to all customers who have signed a license agreement. The use of the source code is limited by the terms of the license agreement.

- Object Code Only - The product function is available to the customer in machine-executable code only. Source materials have the security classification "Confidential - Restricted".
- Partial OCO - The product function contains a combination of elements with OCO, Restricted Materials, or Non-Restricted Materials distribution classifications.

distribution library. Data sets supplied by the product packager containing one or more products that the user restores to disk for subsequent inclusion in a new system.

Distribution libraries are used as input to the SMP/E GENERATE command or the system generation process to build target libraries for a new system. They are also used by SMP/E for backup when a running system has to be replaced or updated. In SMP/E, these data sets are updated by ACCEPT processing, and are identified with the DISTLIB operand.

distribution process. Produces the final form and distribution of product and service materials.

distribution zone. In SMP/E, a group of VSAM records that describe the SYSMODs and elements in the distribution libraries.

DLIB. Distribution library.

E

Early Support Program (ESP). A procedure for the controlled introduction of hardware and software products into the world-wide marketplace. A product's materials are available to only a selected set of customers.

element. In SMP/E, a macro, module, source module, data element, or element installed in a hierarchical file system (HFS) or Java Archive (JAR) file.

element intersection. The existence of more than one element version in a given system or subsystem. See also *common element*.

element MCS. An MCS used to define a new or replacement element, or to update an existing element.

element names. An element naming structure for z/OS that ensures there will be unique names within the system. This structure further ensures that no two elements have the same name unless they are equivalent or are different element types.

element selection. The process of choosing the appropriate modifications to an element from the SYSMODs selected by SMP/E for APPLY or ACCEPT processing from those that have elements in common.

element version. A specific module, macro, source module, or data element that represents one stage in the evolution of that element. The element version is identified by the FMID of the SYSMOD that contains the particular element version. Also see *versioned element*.

environment. The functions (FMIDs) that are installed on a particular system or subsystem (SREL).

ESP. Early Support Program.

F

FCS. First customer ship.

FCS PTFs. PTFs that are required to be on a PUT tape that is available concurrently with a new product's RELFILE tape.

feature. A feature is associated with the Software Distribution order number that has a type, model, and feature code. The type and model form a program number that is unique for a given product offering. The feature code identifies a particular deliverable for the given product offering.

feature code. Part of the Software Distribution order number, which is used to order product materials from Software Distribution. The feature code is used for asset registration and billing, and to identify a specific deliverable that may be ordered for a product. These deliverables include machine-readable SYSMODs, source code, documentation, and so on.

first customer ship (FCS). The date of the first customer shipment as stated in a product's availability letter. This FCS date may be for the product's materials to be sent for the start of an ESP (Early Support Program), or the start of GA (General Availability).

FMID. Function modification identifier.

function. In SMP/E, a product (such as a system component or licensed program) that can be optionally installed in a user's system. Functions are identified to SMP/E by the ++FUNCTION statement. Each function must have a unique FMID.

functionally higher SYSMOD. A SYSMOD that uses the function contained in an earlier SYSMOD (called the functionally lower SYSMOD) and contains additional functions as well.

functionally lower SYSMOD. A SYSMOD whose function is also contained in a later SYSMOD (called the functionally higher SYSMOD).

function modification identifier (FMID). The FMID is the SYSMOD ID of a function SYSMOD and identifies the function that currently owns an element.

function SYSMOD. Any SYSMOD identified by the ++FUNCTION statement. The function SYSMOD is the SMP/E SYSMOD used for product base and dependent functions.

G

GA. General availability.

general availability (GA). The availability of a product's materials for ordering and distribution to all customers. This GA may be preceded by an ESP (Early Support Program).

GENERATE. An SMP/E command used to create a job stream to build a set of target libraries from a set of distribution libraries.

global zone. A collection of records within the SMP/E CSI that contains information defining a common area that SMP/E uses to represent information not specific to a target zone or distribution zone. For example, the global zone is used to describe SYSMODs residing on the PTS. A CSI can contain only one global zone.

H

hardcopy. A printed copy of machine output in a visually-readable form; for example, printed reports, listings, documents, and summaries. Contrast with *softcopy*.

higher functional level. An element version that contains all of the functions of all other relevant versions of that element. See *functionally higher SYSMOD*.

I

indirect library. A partitioned data set used to package elements or JCLIN data instead of packaging them inline or in RELFILEs. Indirect libraries can be used if both of these conditions are met:

- The data set contains element replacements or JCLIN data (not element updates).
- Users who will be installing the SYSMOD have access to the data set.

See also *link library* and *text library*.

installation. The actual installation "installs" product "code" in the distribution and target (execution) libraries. This "code" is not necessarily operational when it is "installed". Customization may be required before a product's function can be used or before a product's service is effective. The standard installation method uses SMP/E to install new or replacement product function onto a system or subsystem. The special generation method uses system, subsystem, or product generation procedures plus SMP/E to install new or replacement product function onto a system or subsystem.

installation process. Installs products, service.

installation verification procedure (IVP). A product should have an IVP. This IVP may then be used by customers and the service and SIPS processes to verify the installation and operation of the product function.

IVP. Installation verification procedure.

J

JCLIN. May be defined as any of the following:

- The SMP/E process of creating or updating the target zone using JCLIN input data.
- The data set that contains the Stage 1 output from a system, subsystem or product generation, used by SMP/E to update or create the target zone.
- The SMP/E JCLIN command used to read in the JCLIN data.
- The ++JCLIN Statement in a SYSMOD that enables SMP/E to perform the target zone updates during APPLY processing.

JCLIN data. The JCL statements associated with the ++JCLIN statement or saved in the SMPJCLIN data set. They are used by SMP/E to update the target zone when the SYSMOD is applied. Optionally, JCLIN data can be used by SMP/E to update the distribution zone when the SYSMOD is accepted.

L

LCG. Local control group.

licensed program (LP). A separately priced program and its associated materials that bear an copyright and are offered to customers under the terms and conditions of an agreement.

link library (LKLIB). A data set that contains link-edited object modules. It is used as an "indirect library" when the object modules are provided in partitioned data sets rather than inline or in relative files.

LKLIB. Link library.

load module. A computer program in a form suitable for loading into main storage for execution. It is usually the output of a linkage editor.

lower functional level. An element version that is contained in a later element version. See *functionally lower SYSMOD*.

LP. Licensed program.

M

machine-readable information (MRI). One or more files that can be electronically distributed, manipulated, and printed by a user.

machine-readable material (MRM). Product materials that are machine-readable (for example, the product RELFILE tape).

macro. An instruction in a source language that is to be replaced by a defined sequence of instructions in the same source language.

MCS. Modification control statement.

media feature code. See *feature code*.

modification control statement (MCS). An SMP/E control statement used to package a SYSMOD. These statements describe the elements of a program and the relationships that program has with other programs that may be installed on the same system.

modification level. A modification level of a function is an additional version or release qualifier used by some products. This modification level qualifier has also been called a "point release". This modification level of a product's function is essentially another "release" of a product version.

module. An element that is discrete and identifiable with respect to compiling, combining with other units, and loading (for example, the output from a compiler or assembler). Synonym for *object module* or *single-module load module*.

MRI. Machine-readable information.

MRM. Machine-readable material.

N

naming conventions. Naming conventions used by the product process functions (development, service, SIPS, and distribution). These naming conventions include:

- COMP ID
- SMP/E SYSMOD names
- Element names (modules, macros, and so on)
- Target and distribution library names.

national language support (NLS). Product support may be required for multiple languages. This support will affect the design, packaging, and service of the product.

negative prerequisite. In SMP/E, a SYSMOD that must not be present in the system in order for another SYSMOD to be successfully installed.

NLS. National language support.

non-restricted materials. A term of distribution for a product function. Also see *distribution class*.

O

object code only (OCO). A term of distribution for a product function. Source materials for OCO products are Confidential-Restricted. Also see *distribution class*.

object deck. Object module input to the linkage editor that is placed in the input stream, in card format.

object module. A module that is the output from a language translator (such as a compiler or assembler). An object module is in relocatable format with machine code that is not executable. Before an object module can be executed, it must be processed by the linkage editor.

When an object module is link-edited, a load module is created. Several modules can be link-edited together to create one load module (for example, as part of SMP/E APPLY processing), or an object module can be link-edited by itself to create a single-module load module (for example, to prepare the module for shipment in RELFILE format or in an LKLIB data set or as part of SMP/E ACCEPT processing). This is also known as an *object deck*.

OCO. Object code only.

offering. See *software offering*.

operating system. See *system* and *base system*.

operating system independent product (OSIP). A product that is supported on more than one operating system and is independent of the function and service level content of any of the operating systems on which the product is supported.

optional materials. Source code and some publications are optional materials for a product.

orderability. The state of whether or not a product function version or release is orderable from an IBM distribution center.

P

package materials. A product's package materials include:

- RELFILE tape (and related items)
- Program directory
- Publications
- Source code
- Element distribution classification data.

Also called the *product package*.

packaging. Adding the appropriate SMP/E modification control statements to elements to create a SYSMOD, then putting the SYSMOD in the proper format on the distribution medium.

IBM software developers must be able to package products that can be handled by service, SIPS, distribution, and installation process automation. This book provides the information required to assist developers in packaging and delivering z/OS products. Product packaging involves product planning, ownership, design, development, build, and service certification.

partitioned data set extended (PDSE). A system-managed data set that contains an indexed directory and members that are similar to the directory and members of partitioned data sets. A PDSE can be used instead of a partitioned data set.

PDSE. Partitioned data set extended.

PE-PTF. Program error PTF.

planning information. The planning information required for a product.

prerequisite. In SMP/E, a SYSMOD that must either be already installed or be installed along with another SYSMOD for that other SYSMOD to be successfully installed.

preventive service. The mass installation of PTFs to avoid rediscoveries of the APAR fixes included in those PTFs.

product. Generally, a software package, such as a licensed program or user application. A product may contain one or more functions and may consist of one or more versions and releases.

product function. See *function* and *function SYSMOD*.

product package. The final form of a product's

product process. The sum of the software processes that are involved with software products and are encompassed within this book.

program directory. The program directory is the installation task documentation. The program directory describes the program materials and program installation.

program error PTF (PE-PTF). A PE-PTF is a PTF that was found to contain an error and is identified on a ++HOLD ERROR statement, along with the APAR that first reported the error.

program object. An executable program stored in a PDSE program library. A program object is similar to a load module, but has fewer restrictions.

program offering. An unwarranted licensed program. Program offerings usually do not have the Programming Service support and usually have one-time pricing.

program packaging. See *packaging*.

program product. Deprecated term for *licensed program*.

product service. See *service SYSMOD*.

program temporary fix (PTF). A temporary solution or bypass of a problem that may affect all users and that was diagnosed as the result of a defect in a current unaltered release of the program. PTFs are identified to SMP/E by the ++PTF statement.

The PTF must be provided for any APAR fix.

PTF. Program temporary fix.

PTF in error. See *program error PTF*.

R

RECEIVE. In SMP/E, the process initiated by the RECEIVE command that reads SYSMODs and stores them on the PTS and CSI global zone for subsequent SMP/E processing.

regression. The condition that occurs when a modification is made to an element by a SYSMOD that is not related to SYSMODs that previously modified the element.

REJECT. In SMP/E, the process initiated by the REJECT command that removes SYSMODs from the PTS and CSI global zone.

relative file (RELFILE) format. A SYSMOD packaging method in which elements and JCLIN data are in sepa-

rate relative files from the MCSs. When SYSMODs are packaged in relative file format, there is a file of MCSs for one or more SYSMODs, and one or more relative files containing unloaded source-code data sets and unloaded link-edited data sets containing executable modules. Relative file format is the typical method used for packaging function SYSMODs.

relative files (RELFILEs). Unloaded files containing modification text and JCL input data associated with a SYSMOD. These files are used to package a SYSMOD in relative file format.

release. A distribution of a new product or new function and APAR fixes for an existing product. Contrast with *version*.

A release is a specific collection of elements that provide a specific level of function. A release can be identified with a base or dependent function. A release is a complete replacement for prior releases of the function. A base or dependent function may have a series of releases identified with it.

RELFILE tape. The RELFILE tape contains one or more product functions in a format that can be installed on a z/OS system or subsystem by SMP/E. It is a multifile, standard label tape containing the SMP/E control statements for the functions and the data libraries for the functions.

replacement modification identifier (RMID). The modification identifier of the last SYSMOD to completely replace a given element.

requisite. A SYSMOD that must be installed before or at the same time as the SYSMOD being processed.

RESTORE. In SMP/E, the process initiated by the RESTORE command that removes applied SYSMODs from the target libraries, target zone and, optionally, the global zone.

RIMs. Related installation materials.

RMID. Replacement modification identifier.

S

SCP. System control program.

service level. The owner of the element (FMID), the last SYSMOD to replace the element (RMID), and all the SYSMODs that have updated the element since it was last replaced (UMIDs).

service order relationship. A relationship among service SYSMODs that is determined by the PRE and SUP operands, and the type of SYSMOD.

ServicePac. A customized service deliverable that is based on customer-supplied system data.

service process. Provides product service in response to internally and externally discovered product problems.

service SYSMOD. Any SYSMOD identified by an ++APAR or ++PTF statement.

shared load module. A load module containing multiple modules, some of which are owned by multiple FMIDs.

shared module. A module that is link-edited into more than one load module or dynamically accessed by more than one load module.

single-CSECT load module. See *single-module load module*.

single-module load module. A load module created by link-editing a single object module by itself (for example, to prepare the module for shipment in RELFILE format or in an LKLIB data set or as part of SMP/E ACCEPT processing).

SMP/E. SMP/E is the IBM product designed to install new function and subsequent service into target libraries and distribution libraries.

SMS. Storage Management Subsystem.

softcopy. Online information. Contrast with *hardcopy*.

Software Distribution. The distribution process function that packages and distributes the final form of product and service package materials.

software offering. A base or dependent function version or release.

source code. Source code includes the element source code and any Optional Source tape. The element source code is the product code in the original coding language as stored in a library system. An Optional Source tape may contain any private macros necessary to assemble source code, and assembler source code.

source material. Includes the source code and source information in publications such as Program Logic Manuals (PLMs), Program Logic Specifications (PLSs), and Product Design Manuals. Also see *distribution class*.

source module. An element containing the source statements that constitute the input to a language translator (such as a compiler or assembler) for a particular translation.

special generation installation method. A product installation technique that uses system, subsystem, or product generation procedures plus SMP/E to install new or replacement product function onto a system or subsystem.

SREL. System release.

standard installation method. A product installation technique that uses the SMP/E RECEIVE, APPLY, and ACCEPT commands to install new or replacement product function onto a system or subsystem.

Storage Management Subsystem (SMS). A DFSMS/z/OS or z/OS or z/OS/DFP facility used to automate and centralize the management of storage. Using SMS, a storage administrator describes data allocation characteristics, performance and availability goals, backup and retention requirements, and storage requirements to the system through data class, storage class, management class, storage group, and ACS routine definitions.

subsystem. A collection of software data sets organized into libraries that form an operational set of product functions. For example, the sets of operational product functions that form subsystems associated with an MVS/370, MVS/XA, MVS/ESA, or z/OS operating system are CICS and CICS-related, IMS and IMS-related, and NCP and NCP-related.

SUP. Service update. SUP is also used for the SMP/E ++VER SUP operand, where SUP means "superseded".

supply code. See the description of feature code under *feature*.

SYSMOD. System modification.

SYSMOD control statement. An SMP/E modification control statement.

SYSMOD down-level control. Ensures that new products are not at a service level lower than their predecessors.

SYSMOD ID. System modification identifier.

SYSMOD packaging. See *packaging*.

SYSMOD relationships. Although individual SYSMODs can be installed independently, certain inter-SYSMOD relations must be observed if the results are to be meaningful. The following SYSMOD relationships are addressed in this publication:

- Archive
- Unconditional
- Conditional
- Hierarchical
- Prerequisite
- Corequisite
- Negative prerequisite
- Delete
- Supersede
- Delete and supersede
- Coexistence

SYSMOD selection. The process of determining which SYSMODs are eligible to be processed by SMP/E.

system. A collection of software data sets organized into libraries that form an operational set of product functions. For z/OS, the set of operational product functions would be those that form an MVS/370, MVS/XA, MVS/ESA, or z/OS operating system. Also see *base system*.

system control program (SCP). Unlicensed base or dependent functions, usually integrated into an operating system.

system modification identifier (SYSMOD ID). The name that SMP/E associates with a system modification. It is specified on the ++APAR, ++FUNCTION, ++PTF, or ++USERMOD statement.

system modification (SYSMOD). A collection of software elements that can be individually distributed and installed. The SYSMOD is the input data to SMP/E that defines the introduction, replacement, or update of product function elements for SMP/E processing into target libraries and associated distribution libraries.

system release (SREL). A 4-byte value representing a system or subsystem and its release level (for example, Z038 specifies z/OS and C150 specifies CICS).

T

target library. A collection of data sets in which the various parts of an operating system are stored. This is sometimes called a system library. Target libraries contain the executable code that constitutes the running system. In SMP/E, these data sets are updated by APPLY processing, and are identified with the SYSLIB operand.

target zone. In SMP/E, a collection of VSAM records in the SMP/E CSI describing the SYSMODs, elements, and load modules in a target library.

test environment. Description of a particular combination of hardware and software that will be used in the testing of a given product function.

test service level. The service level of the product functions supporting a particular test or tests.

text library (TXLIB). A data set containing JCLIN input or replacements for macros, source, or object modules that have not been link-edited. It is used as an "indirect library" when the JCLIN or elements are provided in partitioned data sets rather than inline or in relative files.

transformed data. Data that has been processed by the GIMDTS service routine so that it can be packaged inline in fixed-block 80 records.

TXLIB. Text library.

U

UCLIN. In SMP/E, the command used to initiate changes, through subsequent UCL statements, to the SMP/E database.

UMID. Update modification identifier.

unconditionally coexisting functions. Functions that coexist and must be in the same zone.

update modification identifier (UMID). The modification identifier of the SYSMOD that updated the last replacement of a given module, macro, or source module.

USERMOD. User modification.

user modification (USERMOD). A change constructed by a user to either modify an existing function, add to an existing function, or add a user-defined function. USERMODs are identified to SMP/E by the ++USERMOD statement.

V

version. A separate licensed program that is based on an existing licensed program and that usually has significant new code or new functions. Contrast with *release*.

By associating a version qualifier with the common name, each base function in the set can be uniquely identified. Each version has significant new function as compared to earlier versions. Dependent functions use the same version qualifier as the parent base function.

versioned element. An element that is part of more than one function (for example, one that is part of a base function and a dependent function). See also *element version*.

Z

zone. A partition within an SMPCSI data set.

Publications and Classes

This section tells you more about the SMP/E publications and education on SMP/E that you might find helpful.

Related Publications

The following documents may be helpful.

- *SMP/E User's Guide*, SA22-7773
- *SMP/E Commands*, SA22-7771
- *SMP/E Reference*, SA22-7772

There may be other documents mentioned throughout this book that are not included here. You may look online for those documents too.

Classes and Self-Study Courses for SMP/E Product Packaging

You should contact your IBM rep for course avaiability.

The following courses are recommended for learning SMP/E Product Packaging:

- SMP/E Fundamentals
- Fundamental System Skills in z/OS
- Integrated System Maintenance Using SMP/E
- z/OS Installation and Tailoring

You'll need to go to IBM Global Campus online for current course numbers, schedules, and to find out if they require classroom or can be self-study.

© Copyright IBM Corp. 1986, 2003

Index

Special Characters

++DELETE MCS
  superseding SYSMODs 61
++element MCS
  superseding SYSMODs
    VERSION operand rules 61
++HOLD MCS
  superseding SYSMODs 61
++IF MCS
  examples
    avoiding loss of PTF for previous release 138
    corequisite dependent functions 152
    cross-product prerequisite for service 153
    cross-product service 154
  superseding SYSMODs 61
++JCLIN MCS
  superseding SYSMODs 61
++MOD MCS
  superseding SYSMODs
    CSECT operand rules 61
    LMOD operand 61
++MOVE MCS
  superseding SYSMODs 61
++RENAME MCS
  superseding SYSMODs 61
++VER MCS
  examples
    cross-product service 142
    defining base and dependent functions 144
    defining mutually exclusive functions 155
    deleting a function 143, 151
    establishing the order of dependent functions 151
    fixing an erroneous post-cutoff PTF 140
    replacing a function 139
    superseding an APAR 137
  superseding SYSMODs
    PRE operand rules 61
    REQ operand rules 61
    SUP operand rules 61
    VERSION operand rules 61

A

adding elements 161
ALIAS statement
  link-edit step
    JCLIN processing 90
APAR fixes
  avoiding regression of 137
APAR fixes *(continued)*
  superseding with a PTF 137
automatic library call function
  LIBRARY statement to exclude modules from automatic library search 95
  SYSLIB DD statement in link-edit steps 97

B

base functions
  compared with dependent functions 14
  defining 144
  deleting 139
  deleting and superseding 143, 153
  overview 14
build process 2

C

callable services
  including modules from another product 97
combining elements 161
common elements among SYSMODs
  defining SYSMOD relationships 151
conditional relationships 18
corequisite SYSMODs
  examples
    corequisite dependent functions 152
    cross-product service for a base function with a prerequisite 154
    cross-product service for corequisite base functions 142
    defining a chain of requisite PTFs 148
    erroneous post-cutoff PTF 140
    saving fixes for previous releases 138
cross-product relationships
  corequisite dependent functions 152
  prerequisites for functions 152
  prerequisites for service 153
  service for a base product with a prerequisite 154
  service for corequisite base functions 142
CSECT
  ++MOD MCS operand 61
  specifying order through ORDER statement 90
  superseding SYSMODs 61

D

ddnames
  distribution libraries 95
  SYSPUNCH usage 95

© Copyright IBM Corp. 1986, 2003

DELETE
  example 143
deleting functions
  See also deleting SYSMODs
  explicit deletion 15
deleting SYSMODs
  examples
    base function 139
    dependent function 151
    function with a corequisite 143
dependent functions
  compared with base functions 14
  defining 144
  deleting 151
  establishing the order of 151

E

element updates
  superseding SYSMODs 62
elements
  adding 161
  combining 161
    from different base functions 142, 154
    from different dependent functions 152
  common
    corequisite PTFs for 145
  definition of 1
  migrating
    updating both functions 162
    using a PTF 163
ENTRY statement
  defining a load module entry 90
  for PL/I load modules 90
  JCLIN processing 94
excluding modules from automatic library search 95
EXPAND statement
  JCLIN processing 94
explicitly deleting functions
  packaging options 15

F

FMID
  adding new 28
function
  See function SYSMODs
function SYSMODs
  base functions 14
  choosing between base and dependent functions 14
  installation overview 2
functions
  packaged as function SYSMODs 13

H

hierarchical file system (HFS)
  load modules residing in
    LIBRARYDD comment 98
    SYSLIB DD statement in link-edit steps 98
    SYSLMOD DD statement in link-edit steps 98
high-level languages
  including modules from another product 97

I

IDENTIFY statement
  JCLIN processing 94
implicitly including modules from another product 97
INCLUDE statement
  JCLIN processing 94
  utility input 94
INSERT statement
  JCLIN processing 95
installation
  methods 25
  overview
    functions 2
    service 2
integration process 2

J

JCLIN command
  processing link-edit steps
    creating LMOD entry 96
    creating MOD entry 94
JCLIN data
  superseding SYSMODs 61

L

language-sensitive elements
  packaging examples 156
LIBRARY statement
  JCLIN processing 95
LIBRARYDD comment for pathname in link-edit steps 98
link-edit utility
  parameters recognized by SMP/E 99
LMOD
  superseding SYSMODs
    ++MOD MCS operand rules 61
LMOD entry
  See also load modules
  created by JCLIN 96
load modules
  defining the ENTRY point 90
  hierarchical file system (HFS)
    LIBRARYDD comment in link-edit steps 98
    SYSLIB DD statement in link-edit steps 98

load modules *(continued)*
hierarchical file system (HFS) *(continued)*
SYSLMOD DD statement in link-edit steps 98
including modules from other distribution libraries 90

M

MCS statements
order of 49
migrating elements
updating both functions 162
using a PTF 163
MOD entry
created by JCLIN 94
modules
adding to a new load module 71
adding to an existing load module 71
mutually exclusive functions 155

N

NAME statement
JCLIN processing 96
negative prerequisite SYSMODs
examples 155
NLS (national language support)
packaging options
single base function 121

O

ORDER statement
JCLIN processing 96
using to specify CSECT order 90
OVERLAY statement
JCLIN processing 95

P

packaging
evolution of 1
examples 27
packaging requirements
assessing 5
packaging rules 49
++FUNCTION sysmod_id 28
data element statements 39
PATH
changing for an existing dataset 31
operand for HFS pathname 98
PDS vs. PDS/E
changing for an existing dataset 31
PL/I, using ENTRY statements 90
PRE
++VER MCS operand
superseding SYSMODs 61
prerequisite SYSMODs
examples
defining base and dependent functions 144
defining cross-product prerequisites for functions 152
defining cross-product prerequisites for service 153
defining service for a function 137
defining service that depends on previous service 137
establishing the order of dependent functions 151
product processes
introduction to 1
PTF
common elements 145
cross-product requisites 142, 154
cutoff dates
fixing an erroneous post-cutoff PTF 140
missing ++IF MCS 140
overview 15
requisite chain 145, 148
saving fixes for previous releases 138
superseding an APAR 137

R

record format
changing for an existing dataset 31
regression, avoiding
for mispackaged requisites 140
superseding the lower-level SYSMOD 137
RELFILE tape
construction rules 8
packaging rules 10
REPLACE statement
JCLIN processing 96
REQ
++VER MCS operand
superseding SYSMODs 61
requisite SYSMODs
conditional 18
unconditional 18

S

SMPOBJ
JCLIN processing 95
SUP
++VER MCS operand
superseding SYSMODs 61
superseding SYSMODs
++DELETE MCS 61
++element MCS
VERSION operand rules 61

superseding SYSMODs *(continued)*
++HOLD MCS 61
++IF MCS 61
++JCLIN MCS 61
++MOD MCS
CSECT operand rules 61
LMOD operand 61
++MOVE MCS 61
++RENAME MCS 61
++VER MCS
PRE operand rules 61
REQ operand rules 61
SUP operand rules 61
VERSION operand rules 61
APARs, examples 137
element updates 62
JCLIN data 61
UCLIN data 62
SYSDEFSD DD statement
JCLIN processing 96
SYSLIB DD statement
JCLIN processing 96
link-edit steps 98
PATH operand for HFS pathname 98
SYSLMOD DD statement
JCLIN processing 98
link-edit steps 98
PATH operand for HFS pathname 98
SYSMOD packaging
See also packaging rules
examples 135
SYSMODs
adding new FMIDs 28
definition of 1
evaluating relationships 27
functions 13
overview 13
packaging
definition of 1
PTFs 15
relationships
conditional 18
requisite 18
unconditional 18
rules for packaging 49
SYSPUNCH
JCLIN processing 95

U

UCLIN changes
superseding SYSMODs 62
unconditional relationships 18

V

VERSION
++element MCS operand
superseding SYSMODs 61
++VER MCS operand
superseding SYSMODs 61

IBM®

Program Number: 5647-A01
5655-G44
5694-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.