z/OS

MVS Program Management: User's Guide and Reference

SA22-7643-03

z/OS

MVS Program Management: User's Guide and Reference

SA22-7643-03

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 219.

Fourth Edition, September 2004

This is a revision of SA22-7643-02.

This edition applies to Version 1 Release 6 of z/OS (5694-A01) and to subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this document, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink™ (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: www.ibm.com/servers/eserver/zseries/zos/webqs.html

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 document
- 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.

Contents

Figures

Tables

About this documentation

This documentation is intended to help you learn about and use the end user interfaces provided by the program management component of z/OS. Program management helps you create and execute programs on z/OS. IBM recommends that you use the program management binder to perform these functions. The linkage editor, the batch loader, and the transport utility are older components of program management that, while still supported by IBM, are no longer under development.

- Chapters 1 through 5 of this documentation provide an overview of linking and editing and are recommended reading for all users.
- Chapter 6 provides options that give you more control over the binding process.
- Chapter 7 provides reference material for the binder control statements.
- Chapter 8 provides reference material for interpreting binder output.
- Chapter 9 provides information about binder serviceability aids.
- Appendix A contains information about using the linkage editor and batch loader.
- Appendix B provides a summary of considerations when migrating from the Linkage Editor, load module format, and the Prelinker to Binder and its program format.
- Appendix C provides information about Binder Return Codes.
- Appendix D contains information about Overlay Programs.
- Appendix E contains information on accessibility features in z/OS.
- Notices contains notices, programming information, and trademarks.

Required product knowledge

To use this documentation effectively in an MVS batch environment, you should be familiar with MVS job control language.

Required publications

You should be familiar with the information presented in the following publications:

Publication title	Order number
z/OS MVS JCL Reference	SA22-7597
z/OS MVS JCL User's Guide	SA22-7598

Related publications

The following publications might be helpful:

Publication title	Order number
z/OS MVS Program Management: Advanced Facilities	SA22-7644
z/OS DFSMS: Using Data Sets	SC26-7410
z/OS MVS Diagnosis: Reference	GA22–7588

Referenced publications

Within the text, references are made to other z/OS documentations and documentations for related products. The full titles and order numbers are listed in the following table:

Short title	Publication title	Order number
z/OS MVS Program Management: Advanced Facilities	*z/OS MVS Program Management: Advanced Facilities*	SA22-7644
z/OS DFSMSdfp Utilities	*z/OS DFSMSdfp Utilities*	SC26-7414
z/OS MVS Programming: Assembler Services Guide	*z/OS MVS Programming: Assembler Services Guide*	SA22-7605
z/OS MVS Programming: Authorized Assembler Services Guide	*z/OS MVS Programming: Authorized Assembler Services Guide*	SA22-7608
z/OS MVS Diagnosis: Tools and Service Aids	*z/OS MVS Diagnosis: Tools and Service Aids*	GA22-7589
z/OS MVS JCL User's Guide	*z/OS MVS JCL User's Guide*	SA22-7598
z/OS MVS System Messages, Vol 7 (IEB-IEE)	*z/OS MVS System Messages, Vol 7 (IEB-IEE)*	SA22-7637
z/OS MVS System Messages, Vol 8 (IEF-IGD)	*z/OS MVS System Messages, Vol 8 (IEF-IGD)*	SA22-7638
z/OS C/C++ Programming Guide	*z/OS C/C++ Programming Guide*	SC09-4765
z/OS C/C++ User's Guide	*z/OS C/C++ User's Guide*	SC09-4767
z/OS UNIX System Services Command Reference	*z/OS UNIX System Services Command Reference*	SA22-7802

Notational conventions

A uniform notation describes the syntax of the control statements documented in this publication. This notation is not part of the language; it is merely a way of describing the syntax of the statements. The statement syntax definitions in this documentation use the following conventions:

[ ] Brackets enclose an optional entry. You can, but need not, include the entry. Examples are:
[*length***]**
[MF=E]

| A vertical bar separates alternative entries. When shown inside brackets, you can use one or none of the entries separated by the bar. Examples are:
[REREAD | LEAVE]

[*length* **|** **'S']**

{ } Braces enclose alternative entries. You must use one, and only one, of the entries. Examples are:

BFTEK={S | A}
{K | D}
{*address* **|** **S | O}**

Sometimes alternative entries are shown in a vertical stack of braces. An example is:

```
MACRF={{(R[C | P])}
       {(W[C | P | L])}
       {(R[C],W[C])}}
```

In the preceding example, you must choose only one entry from the vertical stack.

... An ellipsis indicates that the entry immediately preceding the ellipsis can be repeated. For example:

(*dcbaddr***,[(***options***)],. . .)**

‘ ’ A ‘ ’ indicates that a blank (an empty space) must be present before the next parameter.

UPPERCASE BOLDFACE

Uppercase boldface type indicates entries that you must code exactly as shown. These entries consist of keywords and the following punctuation symbols: commas, parentheses, and equal signs. Examples are:

CLOSE , , , ,TYPE=T
MACRF=(PL,PTC)

UNDERSCORED UPPERCASE BOLDFACE

Underscored uppercase boldface type indicates the default used if you do not specify any of the alternatives. Examples are:

[EROPT={ACC | SKP | ABE}]
[BFALN={F | D}]

Lowercase Italic

Lowercase italic type indicates a value to be supplied by you, the user, usually according to specifications and limits described for each parameter. Examples are:

number
image-id
count

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM® messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from the following locations to find IBM message explanations for z/OS® elements and features, z/VM®, VSE/ESA™, and Clusters for AIX® and Linux:

- The Internet. You can access IBM message explanations directly from the LookAt Web site at http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/.

- Your z/OS TSO/E host system. You can install code on your z/OS or z/OS.e systems to access IBM message explanations, using LookAt from a TSO/E command line (for example, TSO/E prompt, ISPF, or z/OS UNIX® System Services running OMVS).
- Your Microsoft® Windows® workstation. You can install code to access IBM message explanations on the *z/OS Collection* (SK3T-4269), using LookAt from a Microsoft Windows command prompt (also known as the DOS command line).
- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example, Internet Explorer for Pocket PCs, Blazer, or Eudora for Palm OS, or Opera for Linux handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from a disk on your *z/OS Collection* (SK3T-4269), or from the LookAt Web site (click **Download**, and select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

z/OS Wizards

If you'd like help with some of those complex tasks that you do infrequently, then check out the z/OS wizards.

Our z/OS wizards are Web-based interactive assistants that ask you a series of questions about the task you want to perform (for example, setting up a Parallel Sysplex). The wizards simplify your planning and configuration needs by exploiting recommended values and by building customized checklists for you to use. For configuration tasks, our wizards also generate outputs like jobs, policies, or parmlib members that you can upload to z/OS and use.

See if our z/OS wizards can work a little magic for you!

`http://www.ibm.com/servers/eserver/zseries/zos/wizards/`

Accessing z/OS licensed documents on the Internet

z/OS licensed documentation is available on the Internet in PDF format at the IBM Resource Link™ Web site at:

`http://www.ibm.com/servers/resourcelink`

Licensed documents are available only to customers with a z/OS license; access to these documents requires an IBM Resource Link user ID and password, and a key code. Based on which offering you chose (ServerPac, CBPDO, SystemPac), information concerning the key code is available in the Installation Guide that is delivered with z/OS and z/OS.e orders as follows:

- *ServerPac Installing Your Order*
- *CBPDO Memo to Users Extension*
- *SystemPac Installation Guide*

To obtain your IBM Resource Link user ID and password, log on to:

`http://www.ibm.com/servers/resourcelink`

To register for access to the z/OS licensed documents:

1. Sign in to Resource Link using your Resource Link user ID and password.

2. Select **User Profiles** located on the left-hand navigation bar.

Note: You cannot access the z/OS licensed documents unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

Printed licensed documents are not available from IBM.

You can use the PDF format on either **z/OS Licensed Product Library CD-ROM** or IBM Resource Link to print licensed documents.

Where to find more information

Where necessary, this document references information in other documents, using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and Documentation APARs for z/OS and z/OS.e, see the online document at:

`publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS`.

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

Summary of changes

Summary of changes
for SA22-7643-03
z/OS Version 1 Release 6

The book contains information previously presented in *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643-02, which supports z/OS Version 1 Release 5.

New information

- New keywords supporting 64-bit code and data have been added to the IMPORT statement. See "IMPORT statement" on page 108.
- Program management diagnostic information has been added. See Chapter 9, "Binder serviceability aids," on page 145.

This book contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Starting with z/OS Version 1 Release 2, you may notice changes in the style and structure of some content in this book — for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our books.

Summary of changes
for SA22-7643-02
z/OS Version 1 Release 5

The book contains information previously presented in *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643-01, which supports z/OS Version 1 Release 3.

New information

- **TRAP=** is a new binder invocation option to control error diagnosis and recovery. See "TRAP: Error recovery" on page 92.

This document contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.

Summary of changes
for SA22-7643-01
as Updated June 2002

The book contains information previously presented in *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643, which supports z/OS Version 1 Release 3.

New information

- "Unsupported input module formats and contents" on page 157

Changed information

- “COMPAT: Binder level option” on page 74
- “EDIT: Edit option” on page 78
- “Binder syntax conventions” on page 97
- “AUTOCALL statement” on page 102
- “INCLUDE statement” on page 109
- “AMODE and RMODE differences” on page 157

Summary of changes
for SA22-7643-00
z/OS Version 1 Release 3

The book contains information previously presented in *z/OS DFSMS Program Management*, SC27-1130.

The contents of *z/OS DFSMS Program Management*, SC27-1130, have been divided and are now contained in the following new publications:

- *z/OS MVS Program Management: User's Guide and Reference* (SA22-7643) – Contains information about the end user interfaces provided by the program management component of z/OS. It presents an overview of the components, describes how to create executable programs from source modules, and describes how to use the binder (starting, defining input, editing data, options, control statements), the linkage editor, and the batch loader.
- *z/OS MVS Program Management: Advanced Facilities* (SA22-7644) – Contains the programming interfaces provided by the program management component of z/OS. It describes the binder application programming interface, macros, user exits, and buffer formats.

An appendix with z/OS product accessibility information has been added.

This book contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.

Chapter 1. Introduction

z/OS provides program management services that let you create, load, modify, list, read, and copy executable programs. With the program management binder, you can create executable modules in either of two formats and store them (depending on the format) in PDS or PDSE libraries, or in z/OS UNIX files. The two types of executable modules are load modules and program objects and may collectively be referred to as 'program modules'. Of these two formats, program objects are the newer. Program objects remove many of the restrictions of the load module format and support new functionality. You can use the z/OS loader to load saved program modules into virtual memory for execution. You can also use the program management binder to build and execute a program in virtual storage in a single step (with some restrictions).

z/OS continues to support the older linkage editor and batch loader programs. However, the program management binder is a functional replacement for these older programs and has many additional enhancements. Because subsequent releases of z/OS might not support these components, IBM strongly recommends you use the binder exclusively. In addition, the program management binder is a functional replacement for the Language Environment prelinker, although z/OS continues to support the use of the prelinker as a separate intermediate step between compilation and binding for the relevant language translators.

This chapter contains an overview of the services provided by each program management component. It also lists other z/OS programs that support program management tasks.

z/OS Program Management components

Although program management components provide many services, they are used primarily to convert object modules into executable programs, store them in program libraries, and load them into virtual storage for execution.

You can use the program management binder and program management loader to perform these tasks. These components can also be used in conjunction with the linkage editor. A load module produced by the linkage editor can be accepted as input by the binder or can be loaded into storage for execution by the program management binder. The linkage editor can also process load modules produced by the binder.

Figure 1 on page 2 shows how the program management components work together and how each one is used to prepare an executable program.

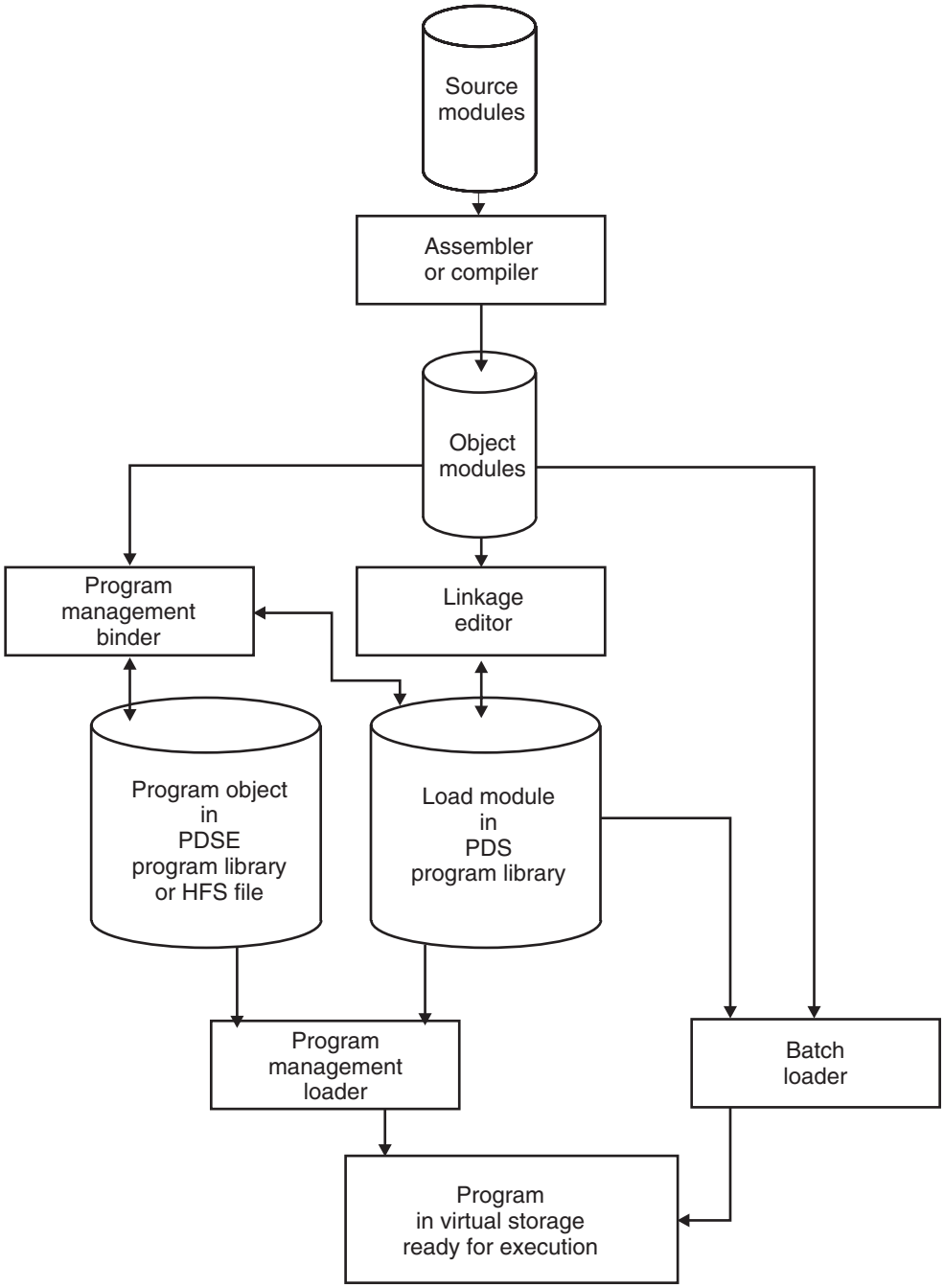


Figure 1. Using Program Management components to create and load programs

The binder

The binder converts the output of language translators and compilers into an executable program unit that can either be read directly into virtual storage for execution or stored in a program library.

Binding program modules

You can use the binder to:

- Convert object or load modules, or program objects, into a program object and store the program object in a partitioned data set extended (PDSE) program library or in a z/OS UNIX file.
- Convert object or load modules, or program objects, into a load module and store the load module in a partitioned data set (PDS) program library. This is equivalent to what the linkage editor can do with object and load modules.
- Convert object or load modules, or program objects, into an executable program in virtual storage and execute the program. This is equivalent to what the batch loader can do with object and load modules.

The binder processes object modules, load modules and program objects, *link-editing* or *binding* multiple modules into a single load module or program object. Control statements specify how to combine the input into one or more load modules or program objects with contiguous virtual storage addresses. Each object module can be processed separately by the binder, so that only the modules that have been modified need to be recompiled or reassembled. The binder can create programs to be loaded into either 24-bit address or 31-bit address storage (for example, RMODE=24 or RMODE=ANY). Beginning with z/OS V1R3, the binder can create programs which execute in 64-bit addressing mode (including support for 8-byte address constants). The binder can also create overlay load modules or program objects (see Appendix D, “Designing and specifying overlay programs,” on page 193). Programs can be stored in program libraries and later brought into virtual storage by the program management loader.

The binder can also combine basic linking and loading services into a single job step. It can read object modules, load modules and program objects from program libraries into virtual storage, relocate the address constants, and pass control directly to the program upon completion. When invoked in this way, the binder does not store any of its output in program libraries after preparing it for execution. Like the batch loader, you can use the binder for high-performance loading of modules that do not need to be stored in a program library.

Enhancements to the binder

The binder also provides the following enhancements compared to the linkage editor:

- Support for single and multi-segment program objects
- Support for a new object module format GOFF
- Easing or elimination of many linkage editor restrictions
- Application programming interface for binding programs
- Increased usability

Program objects: Depending on the library type specified by SYSLMOD, the binder creates either program objects or load modules. Program objects include and extend the functions of load modules. They are stored in partitioned data set extended (PDSE) program libraries or z/OS UNIX files instead of partitioned data set program libraries and have fewer restrictions than load modules. For example, a program object can have a text size of up to 1 gigabyte, whereas the text size of a load module is limited to 16 MB. The block size of a program object is also fixed, eliminating the need to reblock when you copy programs between devices. You can use IEBCOPY to convert between program objects and load modules, as described in “Using utilities for Program Management” on page 6.

Program objects now support an unlimited number of data *classes*, representing multiple text classes, additional control information and user or compiler-specified data known as ADATA. Program text, the instructions and data that constitute the executable portion of the module, can be divided into class segments, each of

which can be loaded into separate storage locations. *Associated Data (ADATA)* is information about the module that is created by the language translator but not required for linking, loading, or execution. Virtually any type of data that is associated with a module or its constituent sections can be saved in a program object. Some restrictions apply.

New object module support: The binder supports a modified extended object module, produced by the COBOL, C and C++ compilers, and a new object module format introduced in a previous release, called *Generalized Object File Format* (GOFF). The extended object module format (XOBJ), allows C, C++ and COBOL programmers to use long external names. The GOFF format (currently produced by the High Level Assembler and the IBM C and C++ compilers) supports long names, multipart modules and ADATA.

Additionally, the binder supports C reentrant modules, dynamic linking, and dynamic link libraries. All object module formats can be stored as sequential files, as members of PDS or PDSE libraries or members of z/OS UNIX archive libraries.

Fewer restrictions: The binder and program objects ease or eliminate many restrictions of the linkage editor and load modules. The linkage editor limited aliases to 64 and external names to 32767. With the binder, the number of aliases and external names for programs stored in a PDSE or z/OS UNIX file is limited only by the space available to store them.

For program objects, external names (those entry points in one section that can be referenced from another section or module or from the operating system) can be up to 32767 bytes in length. Long names can be used for section names, external labels and references, pseudoregisters and common areas, and (limited to 1024 bytes) aliases and alternate entry points for the module. Primary or member names are still limited to eight bytes, however, as are member names appearing in JCL or system macros. For z/OS UNIX-resident program objects, z/OS UNIX name length restrictions apply.

Application Programming Interface: The binder also provides the ability for programs to invoke the binder and request services individually. Binder services can be invoked directly, allowing your programs to access, update, and print the contents of load modules and program objects. For specific information on using the binder application programming interface, see *z/OS MVS Program Management: Advanced Facilities*.

Usability improvements: The binder provides other usability improvements over the linkage editor and batch loader. Messages and diagnostics have been enhanced, producing diagnostic output that is more detailed and easier to understand than the output of the linkage editor. Binder listings are also improved, printing out more complete information about the run that produced a module, including enhancements to the module map and cross reference table and a summary of the data sets used.

There have also been usability improvements (from the linkage editor) in the binder processing options and attributes. A replaceable CSECT in the binder allows the system programmer to establish default options and attributes for the system or installation. In addition, a new SETOPT binder control statement allows users to vary attributes by module when the binder is creating multiple load modules or program objects.

The Program Management loader

The program management loader increases the services of the program fetch component by adding support for loading program objects. The program management loader reads both program objects and load modules into virtual storage and prepares them for execution. It relocates any address constants in the program to point to the appropriate areas in virtual storage and supports 24-bit, 31-bit, and 64-bit addressing ranges.

All program objects loaded from a PDSE are page-mapped into virtual storage. z/OS z/OS UNIX System Services are called to load a program object from a z/OS UNIX file. When loading program objects from a PDSE, the loader selects a loading mode based on the module characteristics and parameters specified to the binder when you created the program object. You can influence the mode with the binder FETCHOPT parameter. The FETCHOPT parameter allows you to select whether the program is completely preloaded and relocated before execution, or whether pages of the program can be read into virtual storage and relocated only when they are referenced during execution. (See “FETCHOPT: Fetching mode option” on page 80 for more information on the FETCHOPT parameter.)

The linkage editor

The linkage editor is a processing program that accepts object modules, load modules, control statements, and options as input. It combines these modules, according to the requirements defined by the control statements and options, into a single output load module that can be stored in a partitioned data set program library and loaded into storage for execution by the program management loader. The linkage editor also provides other processing and service facilities, including creating overlay programs, aiding program modification, and building and editing system libraries. It supports addressing and residence mode attributes in both 24- and 31-bit addressing ranges. It does not support program objects or the (GOFF) object format.

All of the services of the linkage editor can be performed by the binder.

The batch loader

The batch loader combines the basic editing and loading services (that can also be provided by the linkage editor and program fetch) into one job step. The batch loader accepts object modules and load modules, and loads them into virtual storage for execution. Unlike the binder and linkage editor, the batch loader does not produce load modules that can be stored in program libraries. The batch loader prepares the executable program in storage and passes control to it directly. The batch loader cannot accept program objects, GOFF object modules, or control statements as input.

The batch loader provides high performance link-loading of programs that require only basic linking and loading, and can be used if the program only requires listing control or other processing options. Because of its limited options and ability to process a job in one job step, the batch loader only requires about half the combined linking and loading time of the linkage editor and program fetch.

Batch loader processing is performed in a *load* step, which is equivalent to the *link-edit* and *go* steps of the binder or linkage editor. The batch loader can be used for both compile-load and load jobs. It can include modules from a call library (SYSLIB), the link pack area (LPA), or both. The batch loader resolves external references between program modules and deletes duplicate copies of program

modules. It also relocates all address constants so that control can be passed directly to the assigned entry point in virtual storage.

Like the other program management components, the batch loader supports addressing and residence mode attributes in 24-bit and 31-bit bit addressing ranges. The batch loader program is reenterable and therefore can reside in the resident link pack area.

Except for the processing of in-storage object modules, all of the services of the batch loader can be performed by the binder.

Using utilities for Program Management

z/OS provides utility programs to help you manipulate data and data sets. The IEBCOPY, IEHPROGM, and IEHLIST utilities can be used to support program management tasks as described in this section. Information on using these utilities is found in *z/OS DFSMSdfp Utilities*.

Unix System Services commands **cp** and **mv** and TSO commands OGET and OPUT can be used to convert between program modules in a PDS or PDSE and program objects in a USS file system. See *z/OS UNIX System Services Command Reference* for more information.

The binder transport utility (IEWTPORT) is used to convert a program object into a transportable program file and vice versa. For more information on using the transport utility (IEWTPORT), see *z/OS MVS Program Management: Advanced Facilities*.

IEBCOPY

You can use the IEBCOPY utility program to copy a program module from one program library to another. IEBCOPY can also perform conversions between load modules and program objects. IEBCOPY can be used to copy a program module from a partitioned data set program library to a PDSE program library. IEBCOPY converts the new copy into the format appropriate for the target program library. However, you cannot convert a program object into a load module and store it in a partitioned data set library if the program object exceeds the limitations of load modules (for example, if its length is greater than 16 MB).

The control statement, COPYGRP, allows you to copy a program library member (load module or program object) and all of its aliases, specifying only a single name. Since member and alias names are still limited to eight bytes in IEBCOPY control statements, COPYGRP is required for copying members with long alias names.

You can also use the IEBCOPY utility to alter relocation dictionary (RLD) counts of load modules in place, and to reblock load modules. You do not need to alter RLD counts for program objects, or use the COPYMOD control statement to change the block size of a program object library. The COPYMOD control statement reblocks load modules to a block size best suited for the target device, reducing the time it takes to load a program into virtual storage.

IEHPROGM

You can use the IEHPROGM utility or TSO commands to delete or rename load modules, program objects, or their aliases. If the primary name of a PDSE member is deleted or replaced, the associated aliases are deleted automatically. If the

primary name of a PDS member is deleted or replaced, the aliases are not deleted automatically and continue to point to the original member. Aliases for a deleted load module remain unless you specifically delete or replace them.

IEHLIST

You can use the IEHLIST utility or TSO commands to list entries in the directory of one or more partitioned data sets or PDSE program libraries. IEHLIST can list up to ten partitioned data sets or PDSE directories at a time in an edited or unedited format.

The Program Management transport utility

The program management transport utility (IEWTPORT) provides a method for accessing a program object on systems where program management services (that is, the binder) is not installed. The program object is converted by IEWTPORT into a nonexecutable format. The converted object is called a *transportable program*. The transportable program can be transferred to other systems and processed by programs that understand its internal structure. This structure is documented.

IEWTPORT also converts transportable programs into program object format. Load, bind and execute operations are performed on program objects, not transportable programs.

For information on how to invoke the transport utility and how to access a transportable program, see *z/OS MVS Program Management: Advanced Facilities*.

Using service aids for Program Management

Service aids are programs designed to help you diagnose and repair failures in system or application programs. The AMBLIST and AMASPZAP service aids can be used to perform some program management tasks. Both AMBLIST and AMASPZAP support program objects, long names up to 1024 bytes, and multiple text classes. For details on using these programs, see *z/OS MVS Diagnosis: Tools and Service Aids*.

z/OS MVS Diagnosis: Reference contains additional diagnostic information.

AMBLIST

The AMBLIST service aid prints formatted listings of modules to aid in problem diagnosis.

AMBLIST can be used to provide listings showing:

1. The attributes of program modules
2. The contents of the various classes of data contained in a program module, including SYM records, IDR records, external symbols (ESD entries), text, relocation entries (RLD entries), and ADATA
3. A module map or cross reference for a program module
4. The aliases of a program module, including the attributes of the aliases.

Listings of the modified link pack area (MLPA), fixed link pack area (FLPA), pageable link pack area (PLPA), and their extended areas in virtual storage can be printed together or separately.

AMASPZAP

The AMASPZAP service aid, also called SPZAP or Superzap, dynamically updates or dumps programs and data sets. You can use AMASPZAP to inspect and modify instructions or data in any load module or program object in a program library, to dump a load module or program object in a program library, or to update the system status index in the directory entry for any load module or program object. Load modules can be updated in place; when a program object is updated using AMASPZAP, a new copy of the program object is created.

Program objects: Features and processing characteristics

Program objects remove many of the limitations and restrictions inherent in the old load module format. Following are some of the key features of program objects, as well as considerations for their use.

Program object structure

Program objects have the following structural features:

- Program object design allows for the removal or increase of most size restrictions, including maximum text size (now 1 gigabyte) and number of control sections (now unlimited).
- Because program objects reside exclusively in PDSEs, they can take advantage of that library technology and its many advantages.
- The program object structure is generalized and extendable. It will continue to change as required to support new functions.
- Program objects support long names (up to 32767 bytes).
- Program objects contain many of the same enhancements supported in the Generalized Object File Format (GOFF), which is now being generated by the High Level Assembler and a number of high level languages (as well as the Binder itself). This includes support for C/C++ writeable static.
- Program objects contain multiple ***classes*** of text, distinguished by attributes that control binding and loading characteristics and behavior. Classes are central to C and DLL support.
  - There are two types of classes: text (byte-stream) and nontext (record-like, IDR, ADATA)
  - The separate attributes assigned to each class include:
    - LOAD: the class is brought into memory at the time the module is loaded
    - DEFERRED LOAD: The class is prepared for loading, but not instantiated until requested. (Deferred classes are most frequently used by LE for loading multiple dynamically modifiable copies of data.)
    - NOLOAD: The class is not loaded with the program, for example, it is nontext.
    - RMODE 24/ANY: Indicates placement of segments within virtual storage.
  - A ***section*** is the smallest unit that can be manipulated by users (replaced, deleted, ordered). The contribution to a class from a section is called an ***element***; a section may contribute elements to more than one class. Elements (other than parts) may contain entry points.
  - Classes are bound into independently loadable ***segments***. A segment contains classes with compatible attributes. A program object can have multiple segments.

- The loading characteristics of the class (and segment) determine the placement of the segment in virtual storage. Multisegment program objects can be loaded into noncontiguous areas of virtual storage, for example, when bound with the RMODE(SPLIT) option.
- Program objects contain a class of data specifically intended for users to save associated or application data (ADATA). It is not loadable (NOLOAD). This data can be source statements, debugging tables, user information, history data, and documentation. It is accessible via the binder Application Programming Interface defined in *z/OS MVS Program Management: Advanced Facilities*.

Program objects on DASD storage

- Unlike the load module, whose format is documented and universally available, ***the format of the program object is NOT externalized***. The binder API should be used to access program data.
- Consistent with all data in PDSEs, program objects are organized in 4KB blocks, making them accessible by both the binder and loader via DIV (Data in Virtual) access mechanisms. The minimum length of a program object is 4KB.
- When saving a program object in PM1 format, all uninitialized text in a program object (for example, DS space in a program) is written to DASD as binary zeros. DS space is not written to DASD for later program object formats.
- Program objects cannot be in scatter-load format.
- IEBCOPY load/unload functions will process program objects with NO change to the format, that is, it remains the same as it is on DASD.

Residence for and access to program objects

The following describes the program object access modifications and restrictions:

- The program object can be accessed for input using the SAM access method, though this is not recommended. While 4KB blocks will be presented to the user, no description of these blocks will be available. (This access is provided primarily for browse and compare services, where there is no need to interrogate or understand the format of the data.)
- No user can access a PDSE program library directly for output. This function is reserved exclusively for the binder. Services that perform output functions, for example, AMASPZAP, must invoke the binder. Applications can use the binder API to put data into a program object.
- Program objects must reside in either PDSEs or z/OS UNIX files. Data members and program objects may NOT reside in the same PDSE. The PDSE type is determined by the data type on issuance of the first STOW into an empty PDSE.
- There are no “dangling aliases” for program objects in PDSEs. When the primary member name is deleted or replaced, the old aliases are deleted automatically.
- The DCB RECFM field for PDSE program libraries must be specified the same as it is now for PDS program libraries, for example, RECFM=U (undefined record format). While this has no meaning in terms of the actual program object record format, traditionally it has helped to identify program libraries. To promote transparency and usability, this record format will continue to be required as one of the program library indicators for PDSEs as well as PDS’s.

Extensions to the PM loader to support program objects

Most of the loading functions are transparent to the user. The loader will know whether the program being loaded is a load module or a program object by the source data set type. If the program is being loaded from a PDS, it calls

IEWFETCH (now integrated as part of the loader) to do what it has always done. If the program is being loaded from a PDSE, a new routine is called to bring in the program using DIV. The loading is done using special loading techniques that can be influenced by externalized options.

Page mode loading

Program objects can be loaded in **Page Mode**.

- This mode is the default, unless any of the conditions described below under Move Mode exist. Program objects are mapped into virtual storage. If the program object is less than 96K the whole program is preloaded. When over 96K the first 16 pages are preloaded; additional pages are brought in during execution as they are referenced.
- Program objects can be cached in the PDSE hiperspace cache, so frequently referenced pages will be found in cache.
- When the entire module is read in and relocated before execution begins, it is referred to as **Immediate Mode**, a subset of Page Mode.

  An option, FETCHOPT=PRIME, allows you to specify explicitly that the module should be completely relocated before execution. This option only affects Page Mode and forces Immediate Mode. It has the benefit that the loader can immediately release all storage resources that would otherwise be used to contain loader control information (and would usually be held until the module is deleted). It has the disadvantage of bringing in the entire module when it might not be necessary.

Note: Page mode loading is not supported for program objects loaded from z/OS UNIX files.

Move mode loading

Program objects can also be loaded in **Move Mode** from either a PDSE or z/OS UNIX file. This mode is used in those cases where page alignment of virtual storage can not be guaranteed. The entire program is always loaded and relocated before execution. The loader uses Move Mode when:

- A directed load has been requested (for example, the virtual storage address was passed on the LOAD SVC).
- FETCHOPT=PACK was specified at Bind time, forcing Move Mode by requesting that program objects be packed together in virtual storage rather than each be aligned on a page boundary.
- The program object is in overlay format.
- The job step is running V=R.

LLA and checkpoint/restart support for program objects

- ***LLA (Library Lookaside)*** supports both the caching of PDSE program directories and the caching of program objects (loaded from PDSEs), using the same caching algorithms as for load modules. The interfaces to enable LLA are the same as they are today for load modules.
- Programs can be ***Checkpointed and Restarted*** with program objects in the address space ***if*** the PDSE is not open under the user's TCB, (for example, it is OK if PDSEs are JOBLIB, STEPLIB or Linklist). In addition, there must be ***no overlay program objects in the address space*** when a Checkpoint is issued.

Chapter 2. Creating programs from source modules

Program management components process the output of language translators and compilers to produce an executable program unit.

A program can be divided into logical units that perform specific functions. Each of these logical units of code is a *module*. Each module can be written in the symbolic language that best suits its particular function, for example, assembler, C, C++, COBOL, Fortran, or PL/I. Many modules can be bound or link-edited into a single executable program unit. Object modules produced by several different language translators can be merged to form a single program.

Note: This chapter refers to binder processing and output. These concepts apply equally to linkage editor and batch loader processing unless otherwise noted in Appendix A, "Using the linkage editor and batch loader," on page 157. The linkage editor and batch loader cannot process program objects, extended object modules, or GOFF object modules.

Combining modules

Each module of symbolic language code is first assembled or compiled by one of the language translators. The input to a language translator is a *source module*. The output from a language translator is an *object module*. Object modules are relocatable modules of machine code that are not executable, and have one of several formats:

- Traditional object modules (OBJ) produced by most IBM language products and accepted by the binder, linkage editor, and batch loader.
- Extended object modules (XOBJ), for instance those processed by COBOL and C/C++ compilers, are accepted by the Language Environment (LE) prelinker. The binder also accepts XOBJ object files, eliminating the need for the LE prelinker.
- Generalized Object File Format (GOFF) object modules, for example those created by the High Level Assembler and the IBM C/C++ compilers, are accepted only by the binder. GOFF supports long external names up to 32767 bytes, multiple text classes, and embedded ADATA.

Before an object module can be executed, it must be processed by a program management component into executable machine code. The batch loader and the binder can produce executable code directly in virtual storage that executes and is then discarded. The binder and the linkage editor can produce executable code that can be stored in a program library. The binder can produce:

- A program object stored in a partitioned data set extended (PDSE) program library
- A program object stored in a z/OS UNIX System Services (z/OS UNIX) file
- A load module stored in a partitioned data set (PDS) program library.

The linkage editor can only produce load modules stored in a PDS.

You can also use the IEBCOPY utility to convert load modules in a PDS into program objects in a PDSE, or program objects in a PDSE into load modules in a PDS. See "Using utilities for Program Management" on page 6.

Unix System Services commands **cp** and **mv** and TSO commands OGET and OPUT can be used to convert between program modules in a PDS or PDSE and program objects in a USS file system. See *z/OS UNIX System Services Command Reference* for more information.

Program objects and load modules are units of executable machine code in a format that can be loaded into virtual storage and relocated by the program management loader. Collectively, program objects and load modules are referred to as *program modules*. The PDSE and PDS data sets they reside in respectively, are referred to as *program libraries*.

Figure 2 shows the steps required to create an executable program from source modules. The binder API allows you to control specific binding operations. See *z/OS MVS Program Management: Advanced Facilities* for more information about the binder API.

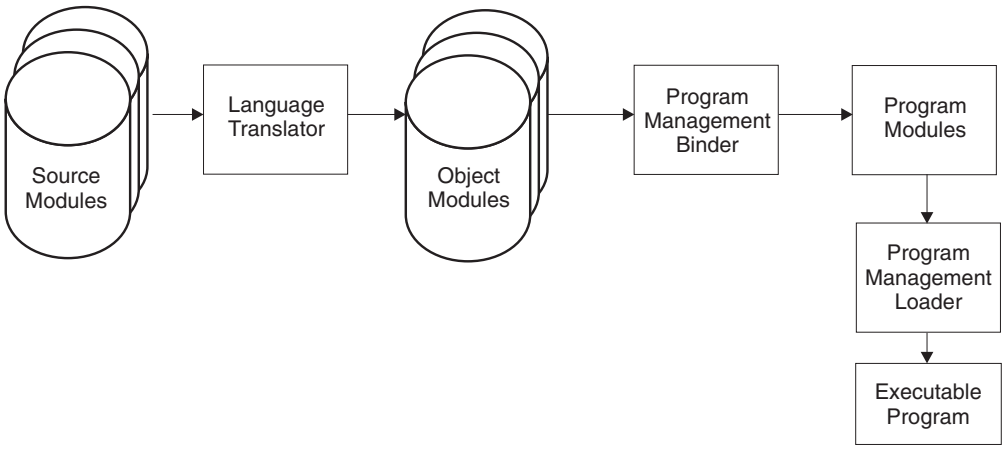


Figure 2. Preparing source modules for execution and executing the program

Sections

Every module is composed of one or more *sections*. A section is a named collection of program object components, called elements, that you can manipulate (for example, order or delete) by that section name during binding. Such manipulation does not affect the integrity of the containing module. The section is a generalization of the traditional object module control section (CSECT) concept.

Sections consist of one or more *elements*, each representing a separate *class* of data. An element does not have a name and cannot be specified on binder control statements. All elements of a section are edited as a unit. If a section is replaced, ordered or aligned, all of its elements are replaced, ordered or aligned. The element represents the cross section of module data identified by a section name and class name.

Classes

Every module is composed of multiple *classes*, each with its own function and format. Some classes represent program *text*, the instructions and data that are loaded into virtual storage during execution. Other classes, such as ESD and RLD, are required for binding and loading the program. Additional classes, such as IDR and ADATA, provide descriptive information about the program module or its individual sections and are of use primarily for maintenance and debugging.

Like sections, classes consist of *elements*. An element is defined by a class name and a section name. Figure 3 illustrates a section/class/element structure.

	Class X	Class Y	Class Z
Section A	Element	Element	Element
Section B	Element	Element	Element

Figure 3. Section/class/element/structure

See “Object and program module structure” on page 16 for the logical structure of elements appearing as one or more classes in a module.

Each element in the class represents the contribution of a single section to that class. The sequence of elements within the class is the same as the sequence of the sections within the module, specified on either the ORDER control statement or the ORDERS API function.

Classes are identified by *class name*. Unlike section names, which are assigned by the source language programmer, class names are normally assigned by an IBM compiler or binder. Class names are a maximum of 16 bytes in length. Binder-defined class names begin with “B_”. Compiler-defined class names begin with “C_”. User-defined class names should not use these prefixes and should be no more than 14 characters long. Class names are not normally required on binder control statements, but can appear in listings and diagnostics. Each separately named class has a specified or an implied set of binding and loading attributes.

Note: The class concept is new with the binder, although several fixed classes (ESD, RLD, TEXT, IDR and SYM) were implicit in the old binding products.

Common areas

A *common area* is a data-only section that can be shared by multiple sections within the module. Common areas can have a name, and if unnamed a name consisting of a single blank will be assumed. The only supported text class for common areas is B_TEXT. If no identically-named CSECT is present, the storage allocated to the COMMON is determined by the longest COMMON definition.

Common areas provide shared space in the module text for data, not instructions. Common areas cannot have initial data values; however, if both a section (CSECT) and common area of the same name are present in the module, the CSECT will initialize the COMMON area. Note that such a CSECT must be at least as long as the longest COMMON definition.

Common areas are normally located at the end (highest virtual address) of the module, but can be relocated by specifying the common area name in the ORDER control statement. When creating a module in overlay format, if a common area is referenced by sections in different paths then it will be moved to a segment higher in the structure (closer to the root segment) that is common to both paths.

Parts

Certain text classes can be further subdivided into *parts*. Like common areas, named parts can be shared between sections and are defined with the longest

length and most restrictive alignment of all contributing sections. Unlike common areas, they must be defined in classes other than B_TEXT. Initializing data in parts is supported for PO3 and later format program objects.

Parts and common areas cannot share the same storage. While both sharing methods can coexist in the same program module, a single shared data area must use one or the other. Older compilers will continue to use common areas for data sharing, whereas newer compilers will utilize parts.

Note: Parts are not supported by either the linkage editor or batch loader programs.

Pseudoregisters

External Dummy Sections, also called *pseudoregisters*, are varying sized units of program storage that do not occupy space in the load module or program object. External Dummy Sections are defined by compilers, or by the assembler using the DXD instruction, and are shared among all sections in the module in the same way that common areas are shared. The attributes of the single, mapped area represents the cumulative length obtained by assigning each pseudoregister's longest length and most restrictive alignment from all its definitions. Virtual storage for the pseudoregister(s) is not provided in the program module, but is instead obtained during execution, using the aggregate length of all pseudoregisters provided by the linker. The concatenation of all uniquely named pseudoregisters is called the *pseudoregister vector*.

All of the linking products (linkage editor, batch loader, and binder) support pseudoregisters, although the implementations are different. The linkage editor and batch loader process pseudoregisters separate from the other program elements and identify them differently in messages and listings. The binder treats pseudoregisters as parts in a "noload" class, B_PRV, and displays the PRV as it would any other class. As a result, there is no separate "Pseudoregister" section in the binder map.

Note: PRV contents are displayed as text class B_PRV. Even though B_PRV is listed as a text class, no text is ever placed in B_PRV by the binder.

Entry points

An *entry point* in a program module is a location that is known by name to the operating system and which can be referenced by or receive control from another module. In PDS and PDSE libraries entry points are represented by directory entries; entry points in z/OS UNIX files are each represented by a file name in the z/OS UNIX directory structure.

There are four types of entry points in program modules:

- **Primary entry point**. This is the point that receives control when the module is invoked by its primary, or member, name. The primary name is the name that was specified on the NAME control statement or the SYSLMOD dd-statement when the module was created.
- **Alternate entry point**. Alternate entry points are locations, other than the primary entry, which can receive control or be referenced from another module. An alternate entry point is defined during binding by use of an ALIAS control statement (or ADDAlias API function) that specifies the name of an external label in the program.

- **True alias**. A true alias is another name associated with the primary entry point. It is also defined with an ALIAS control statement, but is not an external label in the module.
- **Alternate primary**. MVS places certain restrictions on the lengths of member names and aliases. If you specify a name on the NAME control statement that exceeds the 8-byte limitation for member names, the binder will generate an 8-byte primary name and store the specified name as a true alias. This alias is referred to as the *alternate primary* and flagged in the directory entry. The primary entry is also referred to as the *generated primary*.

The linkage editor does not support alternate primaries or any entry point name longer than eight bytes.

The way entry points are represented in the system depends on the type of file in which the module is stored:

- PDSE program libraries support all of the entry point types listed above as directory entries. The primary or generated primary name becomes the member name and is limited to eight bytes. Alternate entry points, true aliases and the alternate primary are stored as aliases and are limited in length to 1024 bytes.
- Partitioned data set (PDS) program libraries support primary entry point, alternate entry point and true alias names up to a maximum of eight bytes. The primary entry point appears as the primary directory entry; aliases and alternate entry points appear as alias directory entries. Alternate primaries are not supported in a PDS.
- z/OS UNIX-resident program objects can contain primary names and true aliases only. All names are limited to 255 bytes, not including the path name. Alternate entry points and alternate primary entry points are not supported. As far as UNIX System Services is concerned, there is no difference between primary names and alias names.

External symbols

Sections can contain symbolic references to locations defined in the same or other sections. These references are called *external references*. External references are normally made by using an *address constant* (adcon). For program objects, the binder supports adcons that are three, four, and eight bytes in length. A symbol referred to by an external reference must be an *external name*, the name of an entry point, or the name of a *pseudoregister*. In modules containing only a single text class, the section (CSECT or common area) name is an implied entry point.

By matching an external reference with an external definition (sometimes called an ‘external label’), the binder resolves references between sections. External references and external labels are called *external symbols*. External symbols are defined in one section and can be referred to in the same section, or from other sections. Figure 4 on page 16 shows how external symbols provide connections between modules.

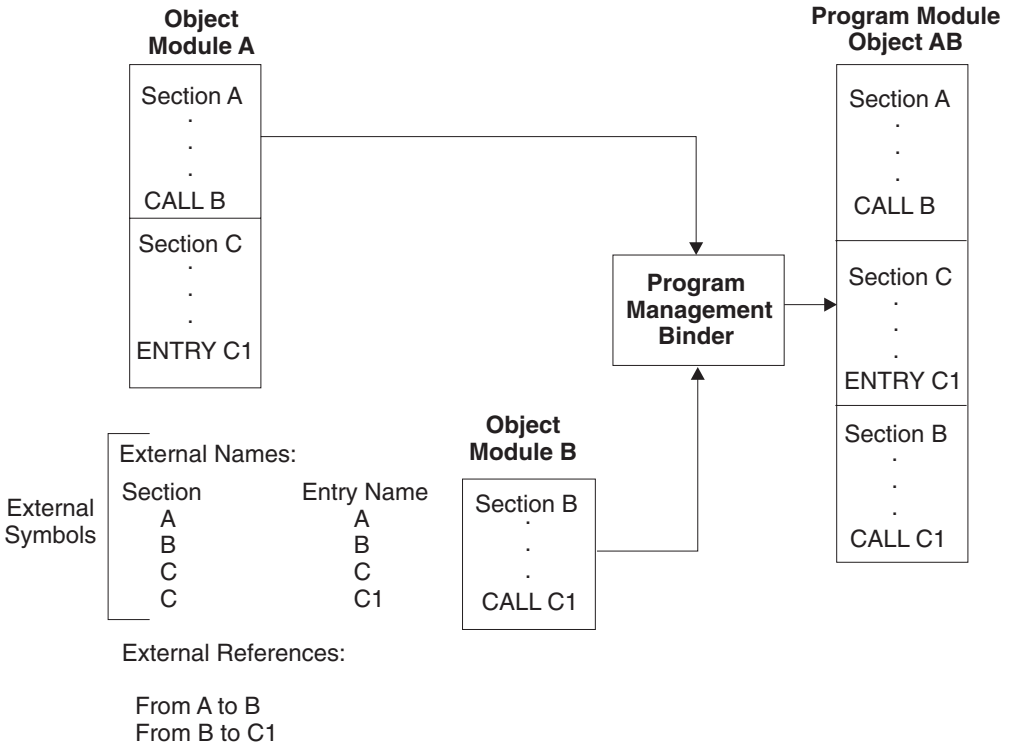


Figure 4. External names and external references

Object and program module structure

Object modules, load modules, and program objects share the same logical structure consisting of:

- Control dictionaries, containing information to resolve symbolic cross-references between sections and to relocate address constants. When a language translator converts source modules into object modules, it generates a control dictionary entry whenever it processes an external symbol, address constant, or section. Most language translators produce two kinds of control dictionaries: an external symbol dictionary (ESD) and a relocation dictionary (RLD).
- Text, containing the instructions and data of the program.
- Identification (IDR) data, containing program control and user-provided information about the modules.
- Associated data (ADATA) for various uses.

Each of these structural elements appears as one or more classes in the module.

A description of the external symbol and relocation dictionaries follows.

External symbol dictionary

The external symbol dictionary (ESD) contains one entry for each external symbol defined or referred to within a module. The dictionary contains an entry for each external reference, entry name, named or unnamed control section, class, blank or named common area, and part or pseudoregister (external dummy section). An

entry name or named control section can be referred to by any control section or separately processed module. An unnamed control section cannot be referred to in this way.

Each entry identifies a symbol or a symbolic reference and gives its location within the module. Each entry in the ESD is classified as one of the following:

External reference
Symbol referenced in the module being processed that is defined as an external name in another separately processed module. The ESD entry specifies the symbol; the location is unknown.

Weak external reference*
External reference that is not resolved by automatic library calls unless an ordinary external reference to the same symbol is found. The ESD entry specifies the symbol; the location is unknown.

External label definition
Name that defines an entry point within a section. For load modules and traditional (OBJ and XOBJ) object modules, an entry point defines an offset within a control section. For program objects and GOFF modules, an entry point defines an offset within an element (and each element is owned by a section). A control section or element may have multiple entry points. The ESD entry specifies the symbol, its location, the addressing mode, and identifies the section or element containing the entry point.

Section definition
In load modules and CSECTs, the symbolic name of a control section. The ESD entry specifies the symbol, the length of the control section, and its location as an offset within the module or program object segment in which the section appears. The location represents the origin, or the first byte, of the control section. This ESD entry also specifies the CSECT addressing mode and residence mode.

In Program Objects, a section is the symbolic name of a collection of elements assigned to one or more classes.

Private code*
Unnamed section. The ESD entry specifies the section length, origin, and can also specify the addressing mode and residence mode of the CSECT. The name field contains blanks.

Blank or named common area*
A section used to reserve a virtual storage area that can be referenced by other modules. The ESD entry specifies the name and length of the area. If there is no name, the name field contains blanks.

Part reference
A reference to a named subdivision of module text that can be shared between referencing sections. Parts might or might not occupy space in the loaded module.

Pseudoregister*
A facility (corresponding to the external dummy section feature of High Level Assembler) that can be used to write reenterable programs. A pseudoregister is part of a dynamically acquired storage area called a pseudoregister vector. The pseudoregister can be of any size or data type. The space for such areas is not reserved in the program module but is acquired during execution. The ESD entry contains the name, length, alignment, and displacement of the pseudoregister.

Element definition
Symbolic name of a class. The ESD entry specifies the attributes of the class. Element definition is supported by GOFF and program objects only.

Tip: The binder requires fewer ESD record types than the linkage editor. Symbol types followed by an asterisk represent variations of the preceding type as they appear in binder listings, GOFF modules, and program objects.

Relocation dictionary

The relocation dictionary (RLD) contains an entry for each address constant that must be modified before a module is executed or requires adjustment during the binding process. The entry specifies both the address constant location within a section and the external symbol used to compute the value of the address constant. (The external symbol can be defined in an ESD entry in another section.)

The binder uses the RLD to adjust (relocate) the address constants for references to other control sections or elements. The RLD is also used to readjust these address constants after the program management loader reads a program object or load module from a program library into virtual storage for execution.

An RLD entry can be one of the following types:

A-con
Non-branch RLD type; in assembler language, DC A(name). The corresponding address constant may contain an offset. A-con's are normally used for branching within a section or for addressing data.

Class address
This type of RLD is supported for PO2 and later format program objects. See "Program object formats" on page 23 for additional information.

Class length
The length of the pseudoregister vector is supported in assembler language by the CXD instruction. In program objects, the length of any class in assembler language uses DC J(classname). For other text classes this RLD type is supported for PO2 and later format program objects. See "Program object formats" on page 23 for additional information.

Loader token
An 8-byte constant which uniquely identifies a specific execution instance of the program (PO3 and later program objects).

Q-con
Q-con type is an offset of the designated symbol from the start of its containing class. In assembler, it is coded as DC Q(name). Q-cons are not relocated during loading. Q-cons designating offsets in class B_PRV are supported for all format modules. For other classes, they are supported for PO2 and later format program objects. See "Program object formats" on page 23 for additional information.

R-con
R-con type is the address of the environment or associated data for a symbol. R-con is supported for program objects in PO3 and later formats. See "Program object formats" on page 23 for additional information.

V-con
V-con is a branch type; in assembler language, DC V(name). V-con's are normally used for branching out of the control section.

Text

Text contains the instructions and the data belonging to the module. The multiclass capability of the binder allows for more than one text class, each of which is loaded into separate storage areas.

Identification data

Identification (IDR) data contains information about the module. The IDR data is not used during program loading and execution. A listing of the IDR data for a module can be obtained by executing the AMBLIST utility.

1. Link-edit or bind identification (IDRB)

   IDRB data identifies the component that created the program module. IDRB data is associated with the entire module never in individual sections.

2. Translator identification data (IDRL)

   IDRL data is produced by the language translator and identifies the compiler or assembler that produced the module or section and the date of compilation.

3. Zap identification data (IDRZ)

   IDRZ data is created by AMASPZAP when it is executed against program modules. It contains a maintenance identifier (such as PTF number) and the date that the maintenance was applied.

4. User identification data (IDRU)

   IDRU data is provided by the user on the IDENTIFY control statement for a program module. It can contain any information pertinent to the associated section. It is created at bind time using the IDENTIFY control statement. See “IDENTIFY statement” on page 106 for more information.

Module attributes

The module attributes include the module entry point designation, module reusability,and the module addressing and residence modes. The primary entry point designation is stored in the END record of an object module. Module attributes for load modules are stored in the directory entry for the partitioned data set member. Module attributes for program objects are stored in the PDSE directory entry and embedded within the program object.

Binder batch processing

This section describes the input and output of the binder and how the binder produces a program object or load module in batch mode.

Input and output

The binder accepts four major types of input:

1. Primary input defined by the SYSLIN DD statement.
2. Additional input specified with the INCLUDE control statement
3. Additional input incorporated by the program management binder from a call library. This input can contain object modules and control statements, load modules, or program objects.
4. Additional input specified as options in the PARM field of the JCL EXEC statement.

Output of the program management binder is of the following types:

1. A program module placed in a program library as a named member, or a program object placed in a z/OS UNIX file. Program objects are stored in PDSE program libraries or z/OS UNIX files. Load modules are stored in partitioned data set program libraries.
2. An executable module loaded into virtual storage.
3. Diagnostic and informational output produced as a sequential data set.

Figure 5 shows how object modules are combined to create a load module.

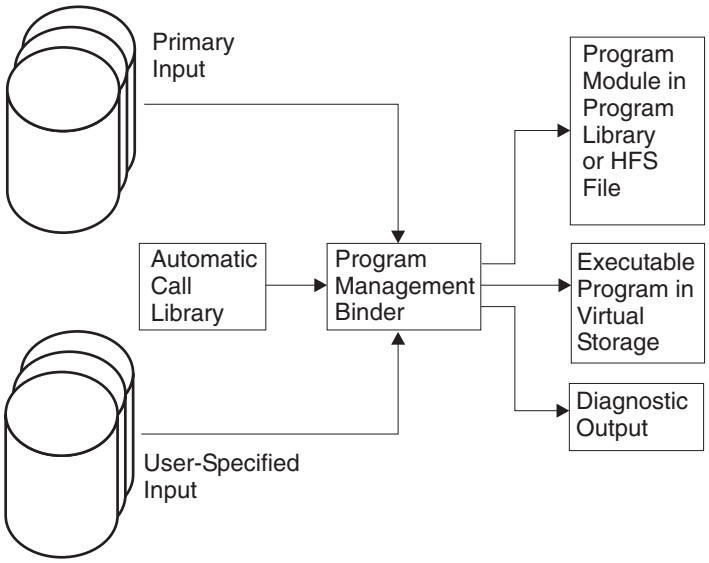


Figure 5. Input and output for the binder

Creating a program module

A program module is composed of all input object modules and program modules processed by the binder or linkage editor. The resultant control dictionaries are collections of all the control dictionaries in the input modules. For load modules, the control dictionaries are merged into a single *composite external symbol dictionary (CESD)* and a single *relocation dictionary (RLD)*. For program objects, the control dictionaries are retained individually. Figure 6 on page 21 shows how multiple input modules are combined into a single program module.

The output module also contains the text from each input module. If the output is a load module, it also contains an end-of-module indicator.

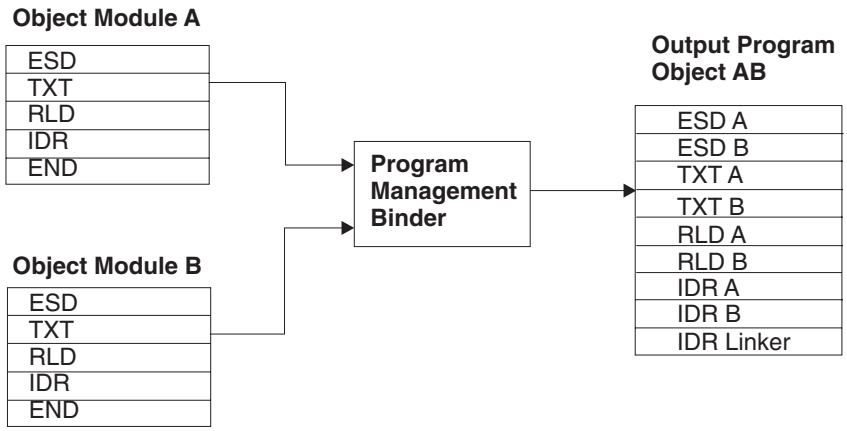


Figure 6. A program object produced by the binder

As the binder processes object and program modules, it assigns relative virtual storage addresses to control sections and resolves references between control sections.

Creating a load module

You can use the binder to create a load module in a PDS. The binder will produce a load module if SYSLMOD is allocated to a PDS. The COMPAT setting has no effect on the decision to produce a load module or a program object. Certain program module contents cannot be saved in a load module and if you have used such features, either the module will be saved with an error indication or you will receive a severe error indicating that the module could not be saved at all. Examples of such features are symbols longer than eight characters or the use of multiple text classes. If you do not use any 64-bit features, then the load module format is compatible across all releases of z/OS and OS/390 and between the binder and the linkage editor. The linkage editor can process load modules produced by the binder and and the binder can process load modules produced by the linkage editor. A load module produced by the binder on z/OS can be loaded and executed on any release of z/OS or OS/390. However, this is not true if the load module has any CSECTs or entry points marked as AMODE(64) or any eight-byte adcons. Such a load module cannot be executed on a release prior to z/OS 1.3 and cannot be processed by the linkage editor.

Creating a program object

You can use the binder to create a program object in a PDSE program library. PDSE program libraries differ in format from PDSE data libraries: Data members, including object modules, and program objects cannot reside in the same library. For the format and content of the PDSE directory entry, see *z/OS MVS Program Management: Advanced Facilities*.

You can also use the binder to create a program object in a z/OS UNIX file. The program object will have the same content as a program object in a PDSE. You can copy a program object from a z/OS UNIX file to a PDSE without loss of information or function. In most cases the same is true for a copy in the other direction; see “Notes on creating a program object in a z/OS UNIX file” on page 22.

Program objects stored in a PDSE library (or z/OS UNIX files) can consist of multiple text classes. At load time, the program management loader will load each text class above or below 16 MB, depending on attributes associated with that text

class. Specifying the RMODE(SPLIT) binder option will cause the module text in B_TEXT to be separated into two classes, B_TEXT24 and B_TEXT31, for loading below and above the line, respectively.

When load modules and old (non-GOFF) object modules are used as inputs to create a program object, the binder converts the old format to the new format by making the following changes:

- Control section names are changed to section names.
- The text of the control section is assigned to class B_TEXT, and an external label entry with the control section's name is associated with the first byte of the element defined by the section name and class B_TEXT, as noted above.
- Pseudoregister items are assigned to class B_PRV.

If you use the capabilities of the High Level Assembler or use the binder RMODE(SPLIT) option to create multipart program objects, certain restrictions apply.

- If the module is the target of a directed load (where the issuer of the LOAD is providing the storage in which to load the module), the two class segments are concatenated and loaded into storage as a single unit.
- All entry points (primary and alternate) must be defined in the same class.
- If parts of the program will reside above 16 MB, then you must ensure that the entire module can execute with AMODE(31) or that linkage between sections on opposite sides of the 16 MB line use BASSM or equivalent instructions to force an AMODE switch when necessary.
- A binder option, HOBSET, will cause the high order bit on V-type address constants to be set according to the addressing mode of the target.
- Overlay format is incompatible with multipart program objects.

If a multipart program object is subsequently loaded via a *directed load* or by the binder, all text classes will be loaded into consecutive storage locations according to the minimum RMODE value for all loaded classes.

Notes on creating a program object in a z/OS UNIX file

Be aware that z/OS UNIX does not support overlay format modules and has more limited alias support (for example, it only supports true aliases and does not support alternate entry points).

You can place a program object in a z/OS UNIX file by specifying the `PATH` parameter on the SYSLMOD DD statement in a batch bind job.

You can also use the binder application programming interface (see *z/OS MVS Program Management: Advanced Facilities*) or the TSO OGET, OGETX, OPUT or OPUTX commands (see *z/OS UNIX System Services Command Reference*) to copy a program object between a PDSE to and a z/OS UNIX file.

When specifying `PATH` in a batch bind job, you can provide either the complete path name or a directory. If the `PATH` parameter designates a directory, you must provide the file name on a NAME statement. The name on the NAME statement must be no longer than 255 bytes.

You can also specify the `PATHOPTS` and `PATHMODE` parameters in the JCL. If you do not, and the JCL designates a directory, the binder assigns attributes for the created file that allow only the file owner to have read, write, and execute authority.

If you specify the PATH parameter for SYSLMOD, the save operation is always processed as though you had specified REPLACE. Also, if you attempt to save a program object to a z/OS UNIX file and do not provide a name through the NAME control statement, the binder does not create a temporary name as it does when you save to a partitioned data set or PDSE under the same circumstance. Refer to the NAME statement under the ″Control Statement Reference″ of this book for a description of said condition.

You can provide an ALIAS control statement to designate the pathname to be used for an alias. The binder appends the path information on the SYSLMOD DD statement to each operand on the ALIAS control statement in order to form each complete alias pathname. The path information on the ALIAS statement must be no longer than 64 bytes per alias. For further information on aliases, see “ALIAS statement” on page 99.

Restrictions:

1. You can execute a program object that resides in a z/OS UNIX file either by using z/OS UNIX commands or through the BPXBATCH facility. You cannot execute such a program object from an MVS batch job using **EXEC PGM=**.
2. z/OS UNIX does not support alternate entry points. All aliases in z/OS UNIX program objects are processed as though they were true aliases.
3. Overlay format modules are not supported in z/OS UNIX files.

Program object formats

There are four program object formats. OS/390 DFSMS Version 1.1 introduced program object format 1 (PO1 format). A PO1 format program object can be executed when using any supported release of OS/390 or z/OS. PO1 is the only format (other than the old load module format) which supports overlay structure within programs. Specifying COMPAT(PM1) or OVLY will cause a module to be saved in PO1 format.

Program object format 2 (PO2 format) was introduced in OS/390 DFSMS Version 1.3. A PO2 format program object can be executed on any currently supported release of OS/390 or z/OS. Specifying COMPAT(PM2) will cause a module to be saved in PO2 format.

Program object format 3 (PO3 format) was introduced in OS/390 DFSMS Version 1.4. All currently supported releases of OS/390 or z/OS also support PO3 format. Specifying COMPAT(PM3) or identifying a currently supported release older than z/OS Version 1.3, such as COMPAT(OSV2R10), will cause a module to be saved in that format.

Program object format 4 (PO4 format) was introduced in z/OS Version 1.3. Only z/OS Version 1.3 and later support PO4 format. Specifying COMPAT(PM4) or identifying ZOSV1R3 or ZOSV1R4 will cause a module to be saved in that format.

A variant of PO4 format is introduced in z/OS Version 1.5. It cannot be rebound or inspected (by Fast Data, the binder API, or AMBLIST) on earlier releases, but it can be loaded and executed on other systems supporting PO4 format. Specifying COMPAT(ZOSV1R5) will cause a module to be saved in that format.

Each program object format introduced support for features not previously available and, except for overlay structure, each format supports all features provided by

earlier formats. By default, the binder will choose the earliest format supporting all of the features being used. See “COMPAT: Binder level option” on page 74 for more information.

Note: As was indicated earlier, the binder also continues to support the old load module format. Note the difference in terminology. A ***load module*** is stored in a standard partitioned data set in a format compatible with older operating systems. A ***program object*** is stored in a PDSE (for example, DSNTYPE=LIBRARY) in one of the formats listed above. The choice between load module and program object for binder output is based solely on the type of data set the program is being stored into.

Binding

Assigning addresses

Each object or load module processed by the binder has an origin that was assigned during assembly, compilation, or a previous execution of the binder or linkage editor. When several modules, each with an independently assigned origin, are to be processed by the binder, the sequence of the addresses is unpredictable. Two input modules can even have the same origin.

Each input module can be made up of one or more sections. To produce an executable program object or load module, the binder assigns relative virtual storage addresses to each section.

The addresses in a program module are consecutive, but are all relative to base zero. When a program is executed, the loading program prepares the module by loading it at a specific virtual storage location and then increasing each address in the program by this base address. Each address constant is also readjusted. This final readjustment is known as *relocation*.

The preceding discussion describes linker actions in processing load modules. When program objects are processed, the output may contain more than one relocatable, loadable segment. In each segment, addresses are relocated during binding relative to a zero base address for each segment; when the segments are loaded, each address constant is relocated relative the the loading address of the segment containing the referenced address. Figure 7 illustrates how multiple segments are created.

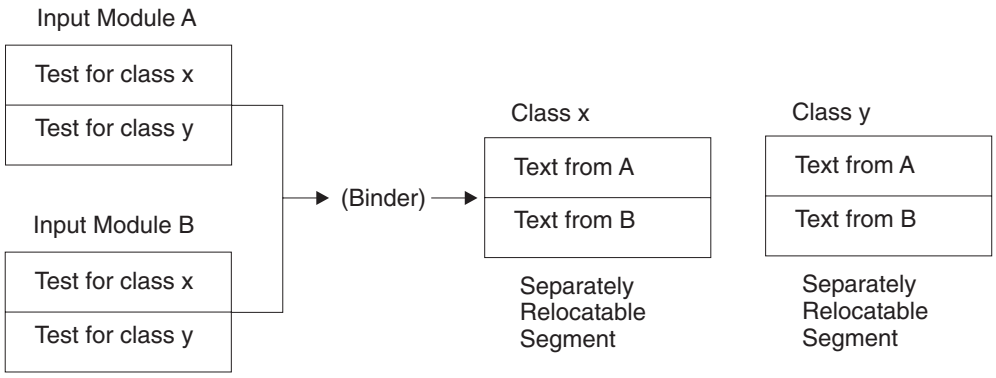


Figure 7. Multiple segments

Resolving external references

The binder resolves module references, matching symbol references to symbol definitions by searching for the external symbol definition in the ESD of each input module. Figure 8 shows the binder matching the external reference to B1 by locating the definition for B1 in the ESD of Module B. In the same way, it matches the external reference to A11 by locating the definition for A11 in the ESD of Module A.

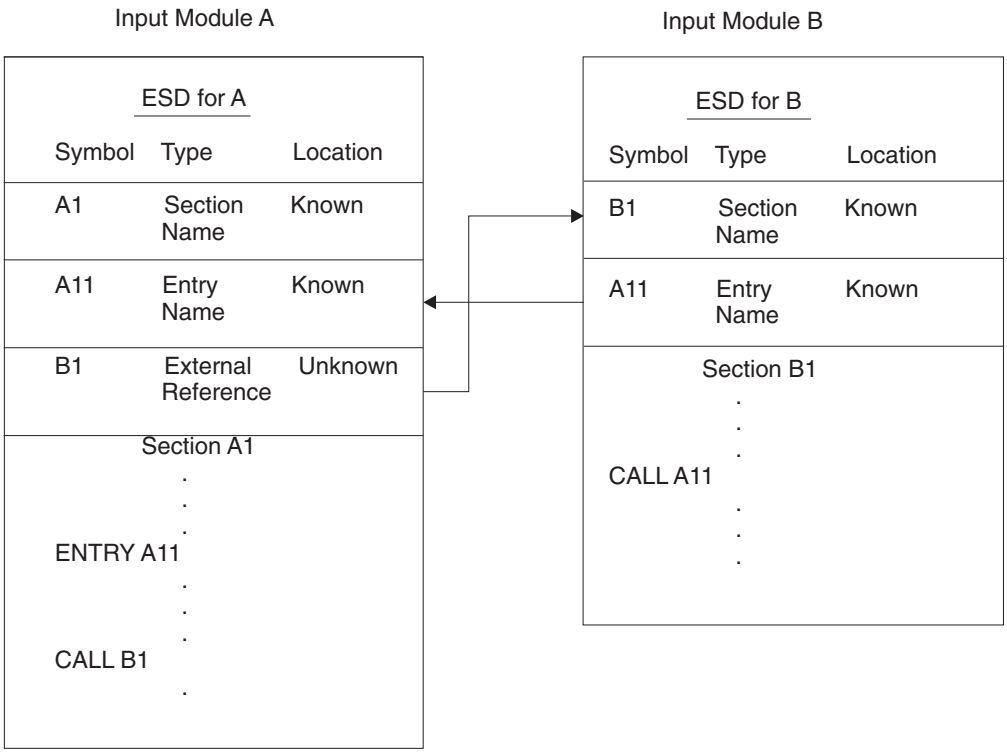


Figure 8. Use of the external symbol dictionary

Note: External names, including section names and entry names, should be one to 32767 bytes in length. No leading or embedded blanks are allowed, nor are the characters outside the range X'41' through X'FE' inclusive. However, the hexadecimal codes X'0E' and X'0F' are recognized as the shift-in and shift-out codes respectively for double-byte character set (DBCS) encoding. All other characters are allowed in any position of the name. Use special characters with caution, because the compilers and assemblers that produce object modules often have a more limited character set and other operating system components may not handle them properly.

Creation of an executable program in virtual storage

The IEWBLDGO entry point of the binder prepares an executable program in virtual storage and passes control to it directly. It combines binding and loading functions into a single step, so it can be used for compile-load-and-go and load-and-go jobs. IEWBLDGO cannot be used to produce a program module in a partitioned data set or a PDSE.

IEWBLDGO cannot be used for programs containing deferred load classes (such as C_WSA). Most XOBJ input to the binder will result in deferred load classes being built.

Addressing and residence modes

A program module has a residence mode assigned to it, and each entry point and alias has an addressing mode assigned to it. You can specify one or both of these modes when creating a program module or you can allow the binder to assign default values. For additional information see “AMODE and RMODE hierarchy” on page 27. The addressing and residence modes must be compatible. The binder, however, allows you to specify them as independent options and validates the combination when the module is saved. See “AMODE and RMODE combinations” on page 27 for information on how the binder resolves addressing and residence modes.

AMODEs and RMODEs can be assigned at assembly or compilation time for inclusion in an object module. AMODE and RMODE values provided to the binder in the ESD data of an object module are retained in the ESD data of the program module (except for overlay programs). Overriding the AMODE and RMODE values in the ESD (see “AMODE and RMODE hierarchy” on page 27) sets the values in the program library directory entry, but does not affect the ESD data.

A special, invalid combination of AMODE(ANY) RMODE(ANY), when appearing in ESD records, is processed as AMODE(MIN). This setting is used by some compilers when creating OBJ-format object modules that do not support AMODE(MIN).

Addressing mode

You assign an addressing mode (AMODE) to indicate which hardware addressing mode is active when the program executes. Addressing modes are:

24 indicates that 24-bit addressing must be in effect.

31 Indicates that 31-bit addressing must be in effect.

ANY Indicates that either 24-bit or 31-bit addressing can be in effect.

64 Indicates that 64-bit addressing can be in effect.

Note: AMODE ANY(64) is not supported.

MIN Requests that the binder assign an AMODE value to the program module. The binder selects the most restrictive AMODE of all control sections in the input to the program module. An AMODE value of **24** is the most restrictive; an AMODE value of **ANY** is the least restrictive.

An AMODE value is provided for each entry point into the program module. The main program AMODE value is stored in the primary directory entry for the program module. Each alias directory entry contains the AMODE value for both the main entry point and the alias or alternate entry point.

Residence mode

You assign a residence mode (RMODE) to specify where the module must be loaded in virtual storage. (Program modules cannot be loaded in data space storage for execution.) Residence modes are:

24 Indicates that the module must reside below the 16-MB virtual storage line (within 24-bit addressable virtual storage).

ANY Indicates that the module might reside anywhere in virtual storage either above or below the 16-MB virtual storage line.

SPLIT Indicates that the module is split into 2 class segments, one to be loaded below 16-MB and one to be loaded above the 16-MB virtual storage line.

The binder places the RMODE value in each directory entry applicable to that program module.

RMODE option and multi-text class modules
Specifying RMODE for a multi-text class module affects only the RMODE of the traditional text class (B_TEXT). The ESD RMODE for a compiler-defined class cannot be overridden.

RMODE(64) Input ESD records may specify RMODE(64). RMODE(64) will be treated as RMODE(ANY) for module loading and execution, although RMODE(64) may appear in the binder MAP output.

AMODE and RMODE hierarchy

The binder uses the following hierarchy to determine the addressing and residence modes of the program module output:

1. Values specified on the binder MODE control statement. See “MODE statement” on page 114 for more information.
2. Values specified in the PARM field of the EXEC statement used to invoke the binder. See “AMODE: Addressing mode option” on page 73 and “RMODE: Residence mode option” on page 89 for more information.
3. Values in the ESD data produced by the AMODE or RMODE assembler statements or by the compiler
4. Default values of AMODE=24 and RMODE=24 when neither AMODE nor RMODE have any specified or derivable values.

AMODE and RMODE combinations

If an AMODE or RMODE value is not specified on a MODE control statement or in the PARM field of an EXEC statement, the binder derives a value based on information in the ESD.

If RMODE is not specified, the module is assigned an RMODE of 24 if either:

- Any section in the module has an RMODE of 24 (note that resident LPA-resident sections resulting from the use of the RES Loader option are not considered when determining RMODE).
- An AMODE of 24 has been specified or defaulted.

Otherwise, the module is assigned an RMODE of ANY. Note that some sections (for example, those resident in the LPA) are not considered when determining RMODE.

If AMODE is not specified, each entry point and alias in the module is assigned the AMODE of that entry point. If the entry point or alias does not correspond to a defined symbol or the symbol does not specify an AMODE, the AMODE of the control section containing the entry point or alias will be used.

If the AMODE of the section containing the entry point is AMODE(MIN) then the entry point is assigned the most restrictive AMODE of all control sections in the input to the program module. Note that the AMODE(MIN) can be in effect due to the conversion of ESD values AMODE(ANY) RMODE(ANY) (see “Addressing and residence modes” on page 26).

AMODE and RMODE validation

The binder validates the AMODE and RMODE combination according to the following table:

	RMODE=24	RMODE=ANY
AMODE=24	valid	invalid
AMODE=31	valid	valid
AMODE=ANY	valid	invalid
AMODE=64	valid	valid

A combination of AMODE=ANY and RMODE=ANY is changed to AMODE=31 and RMODE=ANY unless AMODE=ANY has been directly specified on a control statement or batch parameter. In this case, an error message is issued.

If AMODE is equal to 24 or ANY and RMODE=ANY has been directly specified as a PARM field option or on a control statement, an error message is issued and processing continues.

AMODE and RMODE for overlay programs

All entry points in program modules built in overlay format are assigned an AMODE of 24 and the program modules are assigned an RMODE of 24 regardless of any other values you have specified. RMODE(SPLIT) is not supported for overlay programs.

Module reusability

Reusability is a generic term describing the degree to which a module can be shared, reused or replaced during execution. It incorporates the following attributes:

- Nonreusable. The module is designed for single use only and must be refreshed before it can be reused.
- Serially reusable. The module is designed to be reused and therefore must contain the necessary logic to reset control variables and data areas at entry or exit. A second task cannot enter the module until the first task has finished.
- Reenterable (reentrant). The module is designed for concurrent execution by multiple tasks. If a reenterable module modifies its own data areas or other shared resources in any way, appropriate serialization must be in place to prevent interference between using tasks.
- Refreshable. All or part of the module can be replaced at any time, without notice, by the operating system. Therefore, refreshable modules must not modify themselves in any way.

Unlike AMODE, reusability is an attribute of the entire module, not any particular entry point. It should be chosen based on the operational characteristics of the module and not on the reusability status of individual control sections or data classes.

The linkage editor processed the serially reusable (REUS), reenterable (RENT) and refreshable (REFR) attributes as separate and independent options. The binder, however, treats them as a single, multivalued attribute with an implied hierarchical relationship: “refreshable” implies “reenterable” and “reenterable” implies “serially reusable”. This might result in some confusion for prior linkage editor users who are accustomed to specifying inconsistent combinations of these attributes, such as “REFR,NORENT”. In such situations the binder selects the strongest reusability

attribute among those specified. In addition, unlike the linkage editor, the binder honors any override of reusability specified in the PARM statement.

In order to eliminate such conflicts, specify only a single attribute from the set. Use the keyword(value) form, such as REUS(RENT), rather than keyword-only specifications, such as NORENT or REFR. Note that the refreshable attribute is no longer used by MVS and can be omitted.

Binder extensions supporting the Language Environment

Compatibility with prelinker functions

The binder can directly process XOBJ modules in the format accepted by the IBM Language Environment for MVS & VM prelinker, a utility used as an interim step in the binding of many LE-enabled programs. See *z/OS Language Environment Programming Guide* for additional information.

Added capability in the binder allows for direct processing of XOBJ object modules, obviating the need for the prelinker and simplifying the process for binding such programs. This provides for the creation of rebindable modules, since the binder preserves sufficient information in the saved module to allow the replacement of one or more compilation units.

The binder supports control statements that are functionally equivalent to those offered by the prelinker. The following table shows the relationships between binder and prelinker control statements.

Binder	Prelinker
AUTOCALL	LIBRARY
IMPORT	IMPORT
RENAME	RENAME

Note: Prelinker replacement is supported by the binder only for program objects in PO3 (or later) format. It is not supported for output saved in a load module.

Each XOBJ module will be converted to one or more named or unnamed sections in the program object. The input XOBJ text will be moved to specific binder text classes. The recipe cards in the XOBJ that provide instructions for initializing writable static will be converted into actual initialized text. The following table shows the major classes generated during XOBJ conversion.

Input XOBJ	Class in output program object
reentrant code	C_CODE
writeable static	C_WSA
text in csect STINIT	C_@@STINIT
text in csect DLLI	C_@@DLLI
text in csect PPA2	C_@@PPA2

The binder also creates a table for use by Language Environment run-time routines in class B_LIT. If they are generated, these classes can be seen in the binder map output for section IEWBLIT.

Binder support for DLLs

DLL support in MVS is provided by the z/OS Language Environment component. Only programs that are LE enabled can serve as DLLs or use DLL routines.

The DYNAM(DLL) option controls DLL processing. If DYNAM(DLL) is specified the binder will:

- In some cases, create linkage descriptors in C_WSA
- Process IMPORT control statements
- Build a table of information about imported and exported functions for the use of Language Environment run-time routines. This will appear in the map as class B_IMPEXP.
- Create a side file of IMPORT control statements, corresponding to functions and data being exported by the module being built.

Note: The binder creates sections named IEWBLIT and IEWBCIE. Since this could potentially cause conflict with user-created section names, avoid using section names beginning with the characters IEWB.

For guidance on how to create DLLs and dynamic link libraries, see Building and Using Dynamic Link Libraries (DLLs) in *z/OS Language Environment Programming Guide*.

Chapter 3. Starting the binder

You can invoke the binder as you would any other program: as a job step, a subprogram or a subtask, and as a TSO command. You can execute the binder as a job step by specifying it on an EXEC job control statement in the JCL stream; you can execute it as a subprogram or subtask by using the ATTACH, LINK, LOAD, or XCTL macros. You can execute it under TSO with the LINK and LOADGO commands. This chapter describes these methods of invoking the binder.

Note: This section refers to binder processing and output. These concepts apply equally to linkage editor and batch loader processing unless otherwise noted in Appendix A, "Using the linkage editor and batch loader," on page 157. The linkage editor and batch loader cannot process program objects.

Invoking the binder with JCL

You describe execution of the binder and the data sets used by the binder to the system with job control language (JCL) statements.

This section summarizes those aspects of JCL that apply to the invocation of the binder. The major topics covered are the EXEC statement, the DD statements, and the cataloged procedures for the binder. You should be familiar with JCL as described in *z/OS MVS JCL User's Guide*.

Binder JCL example

Figure 9 on page 32 contains an example of some JCL statements to invoke the binder. You can tailor these statements for your own programming requirements. These statements are similar to the linkage editor JCL statements. In fact, we constructed the example by modifying a set of JCL statements originally used to invoke the linkage editor.

If you need assistance with any of the statements or options, the EXEC statement parameter options are described in Chapter 6, "Binder options reference," on page 67 and the input control statements are described in Chapter 7, "Binder control statement reference," on page 97. The EXEC and DD statements are described in the remainder of this chapter.

```
//LKED     EXEC PGM=IEWL,PARM='XREF,LIST',    IEWL is alias of IEWBLINK
//             REGION=2M,COND=(5,LT,prior-step)
//*
//*        Define secondary input
//*
//SYSLIB   DD  DSN=language.library,DISP=SHR           optional
//PRIVLIB  DD  DSN=private.include.library,DISP=SHR    optional
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))            ignored
//*
//*        Define output module library
//*
//SYSLMOD  DD  DSN=program.library,DISP=SHR            required
//SYSPRINT DD  SYSOUT=*                                required
//SYSTERM  DD  SYSOUT=*                                optional
//*
//*        Define primary input
//*
//SYSLIN   DD  DSN=&&OBJECT,DISP=(MOD,PASS)            required
//         DD  *                           In-stream control statements
  INCLUDE  PRIVLIB(membername)
  ENTRY    entname
  NAME     modname(R)
/*
```

Figure 9. Binder JCL example

EXEC statement

The EXEC statement is the first statement of every job step. For the binder job step, you can specify:

- The program name of the binder
- Binder options passed to the binder program
- Region size requirements for the binder.

EXEC statement—PGM parameter

The PGM parameter on the EXEC statement names the program to be executed. The binder is executed using these program names:

IEWBLINK Binds a program module and stores it in a program library. Alternative names for IEWBLINK are IEWL, LINKEDIT, HEWL, and HEWLH096.

IEWBLDGO Binds a program module, loads it into virtual storage, and executes it. Alternative names for IEWBLDGO are IEWLDRGO, LOADER, and HEWLDRGO.

For example, the following EXEC statement invokes the binder:

```
//LKED       EXEC   PGM=IEWBLINK
```

EXEC statement—PARM field

The EXEC statement can pass various options to the binder using the PARM field. These options perform the following types of services:

- Assigning module attributes that describe the characteristics of the output program module
- Invoking special binder processing services (for example, exclusive call and automatic call)
- Defining the amount of storage to be used by the binder for processing and output program library buffers

- Specifying the kind of output the binder is to produce.

These options can be coded in any order in the PARM field, or can be listed in a data set and included using the OPTIONS keyword.

See Chapter 6, “Binder options reference,” on page 67 for information on individual options.

Preparing the PARM field to invoke the loader: When you invoke the loader, (PGM=IEWBLDGO), both the loader and the loaded program options are specified in the PARM field. The PARM field has this syntax:

,PARM='[*loaderoptions***][/***programoptions***]'**

The loaded program options, if any, must be separated from the loader options by a slash (**/**). If there are no loader options, the program options must begin with a slash. The entire PARM field can be omitted if there are neither loader nor loaded program options. Parameters must be enclosed in single quotation marks when special characters (/ and =) are used.

EXEC statement—REGION parameter

The REGION parameter specifies the maximum amount of storage that can be allocated to satisfy a request for storage made by the binder. You should normally not need to specify this parameter if the installation default region size or system procedures specify enough storage. The recommended minimum region size is 2 MB. While the amount of storage required by the binder is directly related to the number of pieces being bound together (not necessarily the text size itself, but the number of CSECTs, load modules, RLDs, etc. being combined), in most cases 2 MB should be sufficient. The binder executes in 31 bit addressing mode so storage can be obtained from above the line (if available). The recommended values for region size are 2048 KB for program modules with a text size of 1024 KB or less, and twice the text size for program modules with a text size greater than 1024 KB. The binder usually requires a larger region size than the linkage editor. Unlike the linkage editor, the binder does not use temporary disk data sets when virtual storage is exhausted. In addition, the binder can build larger programs than the linkage editor, and so might need more virtual storage.

DD statements

Every data set the binder uses must be described with a DD statement. Each DD statement must have a name, unless data sets are concatenated. The DD statements for data sets the binder requires have preassigned names, those for additional input data sets have names you assign, and those for concatenated data sets (after the first) have no names. When you invoke the binder from another program, you can allocate some or all of the binder's data sets using dynamic allocation instead of JCL.

Binder DD statements

The binder uses eight types of data sets. Some are required, and the DD statements for all but two use the preassigned ddnames shown in Table 1 on page 34. The following descriptions give device and data set information for each binder data set.

Table 1. Binder DDNAMES

Data set	ddname	Required
Primary input data set	SYSLIN	Yes
Options data set	any name	Required when **OPTIONS=***ddname* coded in PARM field of EXEC statement.
Automatic call library	SYSLIB	Only if automatic library call is used
Other include library or sequential data set	any name	Required when referenced on INCLUDE statement
Diagnostic output data set	SYSPRINT SYSLOUT	SYSPRINT is required when using the IEWBLINK entry point.
Output module library	SYSLMOD	Required when using the IEWBLINK entry point.
Alternate output data set	SYSTERM	Only if the TERM option is specified
Output data set for side file (import records used during dynamic binding)	SYSDEFSD	No

SYSLIN DD statement: The SYSLIN DD statement is required. This statement describes the primary input data set, which can be a sequential data set, a partitioned data set member, a PDSE member, an in-stream data set, or a z/OS UNIX file. If it is a z/OS UNIX file, you must specify the PATH parameter.

Each data set in the primary input must contain object modules and control statements, load modules, or program objects. They cannot be mixed within a data set except that control statements can appear before or after an object module in the same data set. Data sets can be concatenated under the SYSLIN DD statement to define the primary input. The binder does not support concatenation of z/OS UNIX files.

"Defining the primary input" on page 44 contains information about input requirements.

The data characteristics vary by data type and are shown in Table 2.

Table 2. SYSLIN data set DCB parameters

LRECL	BLKSIZE	RECFM
80	80	F, FS, OBJ, XOBJ, control statements, and GOFF
80	32720 (maximum size)	FB, FBS OBJ, XOBJ, control statements, and GOFF
84+	32720 (maximum size)	V, VB, GOFF object modules
n/a	32720 (maximum size)	U, load modules
n/a	4096	U, program objects

Options data set: A DD statement defining an options data set is required if the OPTIONS keyword has been included in the PARM field of the EXEC statement. When the OPTIONS keyword is included, some or all of the processing and

attribute options are encoded in a data set instead of in the PARM field. See “OPTIONS: Options option” on page 85 for information on how to code the options data.

The options DD statement is coded using the same ddname as specified on the OPTIONS keyword. The DSNAME parameter references an existing file containing 80-byte records. It may be a sequential data set, a member of a partitioned data set, a z/OS UNIX file sequential data set, or a concatenation of sequential data sets.

SYSLIB DD statement: The SYSLIB DD statement is required if your program has external references that have not been resolved explicitly, unless you have specified the NOCALL option. This DD statement describes the automatic call library, which must reside on a direct access storage device. The data set must be a library and you must not specify member names. You can concatenate any combination of object module libraries and program libraries for the call library. If object module libraries are used, the call library can also contain any control statements other than INCLUDE, LIBRARY, and NAME. If this DD statement specifies a z/OS UNIX file, you can specify either a z/OS UNIX archive library or a `PATH` parameter that designates a directory.

The required data characteristics for object module libraries are the same as those shown in Table 2 on page 34. For program libraries, a record format of U is required. For partitioned data set program libraries, the maximum block size is equal to the maximum for the device used, not the record read. For PDSE program libraries, the block size is 4 KB. You do not specify a value.

The binder does not support z/OS UNIX files as part of a concatenation.

SYSPRINT and SYSLOUT DD statements: If you use IEWBLINK or an alias of IEWBLINK, the SYSPRINT DD statement is required. If you use IEWBLDGO or one of its aliases, you can include a SYSLOUT DD statement, but SYSLOUT is not required. Both SYSPRINT and SYSLOUT describe the diagnostic output data set, which can be a sequential data set assigned to a printer or to a temporary storage device. If a temporary storage device is used, the data records contain an ANSI control character as the first byte.

The usual specification for this data set is SYSOUT=*. The binder uses a logical record length of 121 and a record format of FBA and allows the system to determine an appropriate block size.

Table 3 shows the data set requirements for SYSPRINT and SYSLOUT. Block size is the only information that you can provide.

Table 3. SYSPRINT and SYSLOUT DCB parameters

LRECL	BLKSIZE	RECFM
121	121	FA
121	32670 (maximum size)	FBA
125		VA or VBA

SYSPRINT or SYSLOUT can also be assigned to a z/OS UNIX file. In this case, FILEDATA=TEXT must also be specified.

SYSLMOD DD statement: The following SYSLMOD information applies only to the batch interface of the binder:

- The SYSLMOD DD statement is required. It describes the output program library, which must be a partitioned data set, a PDSE, or a z/OS UNIX file. If it is a z/OS UNIX file, you must specify the `PATH` parameter. z/OS UNIX supports the use of an alternate ddname for SYSLMOD.
- A member name can be specified on the SYSLMOD DD statement. If a member name is specified, it is used only if a name was not specified on a NAME control statement. This member name must conform to the rules for the name on the NAME control statement (see "NAME statement" on page 116).
- If SYSLMOD is referenced by an INCLUDE statement, a member name on the DD statement must be the name of an existing member.

  Note: If you specify the `PATH` parameter on this DD statement, but do not specify `PATHOPTS` or `PATHMODE`, the binder assigns attributes for the created file that allow only the file owner to have read, write, and execute authority.

The following SYSLMOD information applies to both the batch interface and the Application Programming Interface of the binder:

- If the member replaces an identically named member in an existing library, the disposition should be OLD or SHR.
- If the member is added to an existing library, the disposition should be MOD, OLD, or SHR.
- If no library exists and the member is the first added to a new library, the disposition should be NEW or MOD.
- If the member is added to an existing library that can be used concurrently by other users in the system or in other systems sharing the library, the disposition should be SHR.
- Programs which call the binder can specify a different DD name to replace SYSLMOD. All references here to SYSLMOD also apply to that replacement name.
- If SYSLMOD defines a NEW data set, the RLSE subparameter should not be specified since the binder closes the data set after saving each member.
- The binder assigns U for record format for all partitioned data sets, even if you request a different record format. The binder also assigns U for record format for PDSEs if you do not request a record format or if you specify RECFM=U. However, if you specify RECFM with a record format other than U in PDSEs, the binder stops processing and issues an error message.
- The binder always assigns a block size of 4 KB to a program object. Procedures used by the binder to assign block size to a load module are:
  1. If the data set is new:
     a. When the DCBS option is not specified
        – When the data set is created without a block size, the block size is the maximum supported by the access method for that device type.
        – When the data set is created with a block size, the block size specified on the DD statement is used if it is smaller than the maximum block size supported by the device.
        – Certain of the binder options can restrict the blocksize. The block size is:
          - 1KB if the DC option is specified,
          - the value specified on the MAXBLK option,
          - one-half the value specified for *value2* on the SIZE option,

b. When the DCBS option is specified, the block size is the smaller of:
   - The maximum block size for the device
   - The value of the BLKSIZE parameter on the SYSLMOD DD statement
   - The actual output buffer length.

c. The minimum block size is 256 bytes.

2. If the data set already exists:
   - When the DCBS option is not specified, the larger of the existing block size or 256 bytes is used.
   - See “DCBS option” on page 77 for the block size determination when the block size exists and the DCBS option is specified.

In the following example, the SYSLMOD DD statement specifies a permanent partitioned data set library on an IBM 3390 direct access storage device:

```
//SYSLMOD    DD     DSNAME=USER.USERLIB(TAXES),DISP=NEW,UNIT=3390,...
```

The binder assigns a record format of U and a block size of 32760 bytes. However, consider the following example:

```
//LKED       EXEC   PGM=IEWBLINK,PARM='XREF,DCBS'
  :
//SYSLMOD    DD     DSNAME=USER.USERLIB(TAXES),DISP=SHR,UNIT=3390,
//   DCB=BLKSIZE=8000
```

The binder still assigns a record format of U, but the block size is 8000 bytes rather than 32760 bytes because of the use of the DCBS option.

SYSTERM DD statement: The SYSTERM DD statement is optional. It defines a data set for binder messages that supplements the SYSPRINT data set.

SYSTERM output is defined by including a SYSTERM DD statement and specifying TERM in the PARM field of the EXEC statement. SYSTERM output consists of messages that are written to both the SYSTERM and SYSPRINT data sets.

The following example shows the SYSTERM DD statement used to specify the system output unit:

```
//SYSTERM    DD     SYSOUT=A
```

The data set characteristics for SYSTERM (LRECL=80 and RECFM=FB) are supplied by the binder. The block size can be any multiple of 80 bytes acceptable to the hardware. If necessary, the binder modifies the data set characteristics of an existing data set to enforce the LRECL and RECFM values. SYSTERM can also be allocated to a z/OS UNIX file. In this case, FILEDATA=TEXT must also be specified.

SYSDEFSD DD statement: When the DYNAM(DLL) option is used to build a DLL module, a side file might be generated along with it. The side file is saved in the data set represented by the SYSDEFSD ddname. The side file contains the symbols from which other DLLs can import; that is, which symbols the DLL “exports”. Consequently, a side file contains a collection of IMPORT control statements that can be used by other DLLs in order to resolve their own external references during dynamic linking. Starting in z/OS V1.6, side files are enhanced to mark exported function and data which are addressing mode 64 using the types CODE64 and DATA64. If a DLL does not export any symbols, no side file is generated for it.

SYSDEFSD can be a sequential data set, a z/OS UNIX file, a PDS, or a PDSE. If your job binds multiple DLLs and SYSDEFSD represents a sequential data set or a

z/OS UNIX file, the side file records of a given DLL can overwrite or append to the records of a previously saved side file, depending on the DISP or PATHOPTS parameter of your side file ddname.

If SYSDEFSD is a PDS or a PDSE, the binder saves the side file as a member of the indicated partitioned data set. The binder progresses through the following sources until it determines the name to use for the side file:

1. The binder uses the member name specified in the JCL for the SYSDEFSD DD. Note that in this case the side file is treated as a sequential file.
2. If no member was specified, the binder uses the name specified in the NAME control statement for the saved DLL.
3. If there is no NAME control statement, the binder uses the name expressed in the JCL SYSLMOD DD statement.

The SYSDEFSD DD statement is optional. However, when it is absent, the binder issues a warning message if at bind time a module (DLL) generates export records and the DYNAM(DLL) binder option has been specified. Note that the side file can be referred to as the definition side deck by other products.

Table 4 shows the data set requirements for SYSDEFSD.

Table 4. SYSDEFSD DCB parameters

LRECL	BLKSIZE	RECFM
80	32760 (maximum size)	F,FB

Additional DD statements

Each ddname specified on an AUTOCALL, INCLUDE or LIBRARY control statement must be defined with a DD statement. These DD statements describe sequential data sets, partitioned data sets, PDSEs, or z/OS UNIX files.

You specify the ddnames along with any other necessary information. The requirements for these data sets are shown in Table 5.

Table 5. INCLUDE and LIBRARY control statements DCB parameters

Data set contents	LRECL	BLKSIZE	RECFM
Object modules or control statements	80 80	80 32760 (maximum)	F, FS FB, FBS
Load modules	Ignored	Maximum for device, or value specified on the MAXBLK option, whichever is smaller	U
Program objects	Ignored	4096	U

Binder cataloged procedures

The MVS operating system allows you to store job control statements under a unique member name in a procedure library. Such a series of statements is called a *cataloged procedure*. These JCL statements can be recalled at any time to specify the requirements for a job. To request this procedure, place an EXEC statement in the input stream. This EXEC statement specifies the unique member name of the desired procedure.

The specifications in a cataloged procedure can be temporarily overridden, and DD statements can be added. The information that you alter is in effect only for the

duration of the job step; the cataloged procedures are not altered permanently. Any additional DD statements that you supply must follow those that override existing JCL statements in the same procedure step. For more information on using cataloged procedures, see *z/OS MVS JCL User's Guide*.

Two binder cataloged procedures are provided: a single-step procedure that binds the input and produces a program module (LKED procedure), and a two-step procedure that binds the input, produces a program module, and executes that module (LKEDG procedure). Many of the cataloged procedures provided for language translators also contain binder steps. The EXEC and DD statement specifications in these steps are similar to the specifications in the cataloged procedures described in the following paragraphs.

LKED procedure

LKED is a single-step procedure that binds the input, produces a program module, and passes the module to another step in the same job.

```
//LKED  EXEC PGM=HEWLH096,PARM='MSGLEVEL(4),XREF,LIST,LET,NCAL',
//         REGION=2M
//SYSPRINT DD  SYSOUT=A
//SYSLIN   DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(GO),SPACE=(1024,(50,20,1)),
//         UNIT=SYSDA,DISP=(MOD,PASS)
```

Statement description: A description of the statements in the procedure follows:

EXEC
The PARM field specifies the NCAL option. If an automatic call library is used, you must override the NCAL option and add a SYSLIB DD statement.

SYSPRINT
Specifies the SYSOUT class A, which is either a printer or a temporary storage device. If a temporary storage device is used, ANSI control characters accompany the data to be printed.

SYSLIN
The specification of DDNAME=SYSIN allows you to specify any input data as long as it fulfills the requirements for binder input. You must define the input data with a SYSIN DD statement. This data can be either in the input stream or reside in one or more separate data sets.

If the data is in the input stream, use the following DD statement:

```
//LKED.SYSIN   DD     *
```

Place the SYSIN statement following all overriding DD statements for the LKED catalog procedure. The object module decks and control statements should follow the SYSIN statement, with a delimiter statement (/*) at the end of the input.

If the data resides in separate data sets, use the following DD statement:

```
//LKED.SYSIN   DD     (parameters describing the input data set)
```

Place the SYSIN statement following all overriding DD statements for the LKED catalog procedure. Several data sets can be concatenated as described in Chapter 4, "Defining input to the binder," on page 43.

SYSLMOD
Specifies a temporary data set and a general space allocation. The disposition allows the next job step to execute the program module. If the module is to reside permanently in a library, these general specifications must be overridden.

Invoking the LKED procedure: To invoke the LKED procedure, code the following EXEC statement:

```
//stepname   EXEC   LKED
```

The following example shows a sample JCL sequence for using the LKED procedure in one step to bind object modules to produce a program module, then execute the program module in a subsequent step.

```
//LESTEP     EXEC   LKED
  (Overriding and additional DD statements for the LKED step)
//LKED.SYSIN DD     *
  (Object module decks and control statements)
//EXSTEP     EXEC   PGM=*.LESTEP.LKED.SYSLMOD
  (DD statements and data for load module execution)
```

LESTEP invokes the LKED procedure and EXSTEP executes the program module produced by LESTEP.

LKEDG procedure

LKEDG is a two-step procedure that binds the input, produces a program module, and executes that module. The statements in this procedure are shown in the following example. The two procedure steps are named LKED and GO. The specifications in the statements in the LKED step are identical to the specifications in the LKED procedure.

```
//LKED   EXEC PGM=HEWLH096,PARM='MSGLEVEL(4),XREF,LIST,NCAL',
//         REGION=2M
//SYSPRINT DD  SYSOUT=A
//SYSLIN   DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(GO),SPACE=(1024,(50,20,1)),
//         UNIT=SYSDA,DISP=(MOD,PASS)
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED)
```

GO Step

The EXEC statement specifies that the program to be executed is the program module produced in the LKED step of this job. This module was stored in the data set described on the SYSLMOD DD statement in that step. (If a NAME statement was used to specify a member name other than that used on the SYSLMOD statement, use the LKED procedure and provide your own GO step.)

The condition parameter specifies that the execution step is bypassed if the return code issued by the LKED step is greater than 4.

Invoking the LKEDG procedure: To invoke the LKEDG procedure, code the following EXEC statement:

```
//stepname   EXEC   LKEDG
```

The following example shows a sample JCL sequence for using the LKEDG procedure to bind object modules, produce a program module, and execute that module.

```
//TWOSTEP    EXEC   LKEDG
  (Overriding and additional DD statements for the LKED step)
//LKED.SYSIN DD     *
  (Object module decks or control statements, or both)
/*
  (DD statements for the GO step)
//GO.SYSIN   DD      *
  (Data for the GO step)
/*
```

Invoking the binder under TSO

You can invoke the binder under TSO (Time Sharing Option) with the LINK and LOADGO commands. You may also be able to run it from an ISPF foreground panel, and if you want to do all of the allocations yourself, you can use CALL.

The LINK command creates a program module and saves it in either a partitioned data set or PDSE program library.

When using the LINK command to process binder control statements, you must allocate any referenced ddnames before the LINK command is invoked. The binder gives you the capability of including modules and control statements from the automatic call library (SYSLIB) or including program modules from the module output library (SYSLMOD). If you specify SYSLIB or SYSLMOD on an INCLUDE statement but have not allocated data sets to those ddnames, the binder will attempt to process the INCLUDE statement using the data sets indicated on the LIB or LOAD parameters, respectively.

The LOADGO command creates and executes a program module. The module is not saved in a program library. The LOADGO command invokes a prompter that allows you to define any necessary data sets to the system; you can use LOADGO operands to specify the loading options the job requires.

To use the TSO CALL command, you first need to use ALLOCATE to set up file names corresponding to the JCL DD statements described earlier in this chapter. Then, use the following command to invoke the binder:

```
CALL *(IEWL) 'options'
```

See *z/OS TSO/E Command Reference* for the procedures for using these commands.

Invoking the binder from the z/OS UNIX Shell

You can invoke the binder from the z/OS UNIX shell using the c89 command. See *z/OS UNIX System Services Command Reference* for more information.

Invoking the Binder from a program

You can pass control to the binder from a program in one of two ways:

1. As a subprogram, with the execution of a CALL macro instruction (after the execution of a LOAD macro instruction), a LINK macro instruction, or an XCTL macro instruction.
2. As a subtask with the execution of the ATTACH macro instruction.

For addiitonal information, see *z/OS MVS Program Management: Advanced Facilities*.

Chapter 4. Defining input to the binder

Batch input to the binder consists of the primary input data set and additional data sets. You define the primary input data set using job control statements. You can include more modules by specifying additional control statements and by directing the binder to use call libraries.

Input data sets can contain control statements, object modules of any type, load modules and program objects. The following table shows the data set types in which data can reside.

	Sequential data set	**PDS member**	**PDSE member**	**z/OS UNIX file**
Control Statements	X	X	X	X
Object Modules (all types)	X	X	X	X
Load Modules		X		
Program Objects			X	X

A single library member can contain only one program object or load module, but any number of control statements and object modules in combination.

z/OS UNIX files can contain binder input of all types except load modules. You specify z/OS UNIX either by coding the PATH parameter on your JCL or by providing the path name on the INCLUDE or AUTOCALL control statements. See Chapter 7, “Binder control statement reference,” on page 97 for more information. Where sequential processing or archive file access is required, you must include the full file name on the PATH parameter; otherwise, code only the directory name for PATH, omitting the last level of qualification (file name). The file name will be supplied by the binder, either from the INCLUDE statement or from the unresolved reference during autocall.

In addition to the data set type, you must consider how the binder will access the data set. Sequential access requires that a physical sequential data set be specified or that a member name be specified with the library dsname. Partitioned access requires that a partitioned data set, PDSE, z/OS UNIX archive file, or z/OS UNIX directory be specified without an associated member or file name. Access requirements depend on the time that the input is required:

- Primary input is accessed sequentially. Any library in the concatenation must include a member name with the dsname or path.
- Secondary (included) input can be either sequential or partitioned. If partitioned, the member name(s) must be specified on the INCLUDE control statement.
- Autocalled input must be partitioned.

The binder supports mixed concatenations of the above, with the following exceptions:

- You must not mix data set types in a single concatenation. All concatenated data sets must be either partitioned or sequential, not both. A PDS or PDSE member is treated as a sequential data set
- The binder does not support z/OS UNIX files concatenated with other z/OS UNIX files or data sets of any type.

Note: This chapter refers to binder processing and input. These concepts apply equally to linkage editor and batch loader processing unless noted otherwise in Appendix A, “Using the linkage editor and batch loader,” on page 157. The linkage editor and batch loader cannot process program objects, extended object modules, GOFF modules, C370LIBs or z/OS UNIX files.

Defining the primary input

The primary input, required for every binder job step, is defined on a DD statement with the ddname SYSLIN. Primary input can be:

- A sequential data set
- A member of a partitioned data set (PDS)
- A member of a partitioned data set extended (PDSE)
- Concatenated sequential data sets, or members of partitioned data sets or PDSEs, or a combination
- A z/OS UNIX file.

The primary data set can contain object modules, control statements, load modules and program objects. All modules and control statements are processed sequentially and their order determines the order of binder processing. The order of the sections after processing, however, might not match the input sequence.

The following examples show the statements needed to define input to the binder.

Object modules, load modules and program objects

Primary input to the binder can be one or more object modules, load modules or program objects. The modules are created and passed by a previous job step or created in a separate job.

As a member of a partitioned data set or PDSE

You can use a module in a partitioned data set or PDSE as primary input to the binder by specifying its data set name and member name on the SYSLIN DD statement. In the following example, the member named TAXCOMP in the object module library USER.LIBROUT is the primary input. USER.LIBROUT is a cataloged data set:

```
//SYSLIN     DD     DSNAME=USER.LIBROUT(TAXCOMP),DISP=SHR
```

The library member is processed as if it were a sequential data set.

Members of partitioned data sets or PDSEs can be concatenated with other input data sets, as follows:

```
//SYSLIN     DD     DSNAME=USER.OBJMOD,DISP=SHR,...
//           DD     DSNAME=USER.LIBROUT(TAXCOMP),DISP=SHR
```

Library member TAXCOMP is concatenated to data set USER.OBJMOD.

Passed from a previous job step

A module used as input can be passed from a previous job step to a binder job step in the same job (for example, the output from the compiler is direct input to the binder). In the following example, an object module that was created in a previous job step (STEPA) is passed to the binder job step (STEPB):

```
//STEPA      EXEC
//SYSGO      DD     DSNAME=&&OBJECT,DISP=(NEW,PASS),...
   :
//STEPB      EXEC
//SYSLIN     DD     DSNAME=&&OBJECT,DISP=(OLD,DELETE)
```

The temporary data set name &&OBJECT, used in both job steps, identifies the object module as the output of the language processor on the SYSGO DD statement, and as the primary input to the binder on the SYSLIN DD statement.

Created in a separate job

If the only input to the binder is an object module from a previous job, the SYSLIN DD statement contains the information needed to locate the object module. For example:

```
//SYSLIN     DD     DSNAME=USER.OBJMOD,DISP=(OLD,DELETE)
```

Control statements

The primary input data set can consist solely of control statements. When the primary input is control statements, input modules are specified on INCLUDE control statements (see “Secondary (included) input” on page 46). The control statements can be either placed in the input stream or stored in a data set.

In the following example, the primary input consists of control statements in the input stream:

```
//SYSLIN     DD     *
  Binder Control Statements
/*
```

In the next example, the primary input consists of control statements stored in the member INCLUDES in the data set USER.CTLSTMTS:

```
//SYSLIN     DD     DSNAME=USER.CTLSTMTS(INCLUDES),DISP=SHR,...
```

In either case, the control statements can be any of those described in Chapter 7, “Binder control statement reference,” on page 97.

Modules and control statements

The primary input to the binder can contain modules and control statements. The object modules and control statements can be in the same data set or in different data sets, but cannot be mixed in the same data set with load modules or program objects.

If the modules and statements are in the same data set, this data set is specified in the SYSLIN DD statement. If the modules and statements are in different data sets, the data sets are concatenated. The binder accepts concatenated object modules, load modules and program objects as primary input. However, the binder does not support z/OS UNIX files as part of a concatenation. The control statements can be defined either in the input stream or as a separate data set.

Control statements in the input stream

Control statements can be placed in the input stream and concatenated to an object module data set, as follows:

```
//SYSLIN     DD     DSNAME=&&OBJECT,...
//           DD     *
  Binder Control Statements
/*
```

Another method of handling control statements in the input stream is to use the DDNAME parameter, as follows:

```
//SYSLIN     DD     DSNAME=&&OBJECT,...
//           DD     DDNAME=SYSIN
             .
             .
             .
//SYSIN      DD     *
  Binder Control Statements
/*
```

Note: The binder cataloged procedures use DDNAME=SYSIN for the SYSLIN DD statement to specify the primary input data set required.

Control statements in a separate data set

A separate data set that contains control statements can be concatenated to a data set that contains an object module. Control statements for a frequently used procedure (for example, a series of INCLUDE statements) can be stored permanently. In the following example, the members of data set USER.CTLSTMTS contain binder control statements. One of the members is concatenated to data set &&OBJECT.

```
//SYSLIN     DD     DSNAME=&&OBJECT,DISP=(OLD,DELETE),...
//           DD     DSNAME=USER.CTLSTMTS(MEDIA),DISP=SHR,...
```

The control statements in the member named MEDIA of the data set USER.CTLSTMTS are used to structure the resultant module.

Secondary (included) input

The INCLUDE control statement requests that the binder use additional data sets as input. These can be any of the sequential data set types acceptable for primary input.

In addition, INCLUDE can refer to private libraries rather than sequential files. Concatenations must contain only libraries or sequential files (including library members), not both.

The INCLUDE statement specifies the ddname of a DD statement that describes the data set to be used as additional input. If the DD statement describes a library (partitioned data set, PDSE, or z/OS UNIX directory) the INCLUDE statement also contains the name of each member to be used. See “INCLUDE statement” on page 109 for the syntax of the INCLUDE statement.

When an INCLUDE control statement is encountered, the binder processes the module or modules indicated. Figure 10 on page 47 shows the processing of an INCLUDE statement. In the illustration, the primary input data set is a sequential data set named OBJMOD that contains an INCLUDE statement. After processing the included data set, the binder processes the next primary input item. The arrows indicate the flow of processing.

If an included data set also contains an INCLUDE statement, that INCLUDE is processed at the time it is encountered, effectively nesting includes. Any number of nested INCLUDE statements are possible with the binder. Figure 10 on page 47 demonstrates the flow of processing for single INCLUDE statements. Note that the binder returns to the Include module after processing the included module whereas the linkage editor does not.

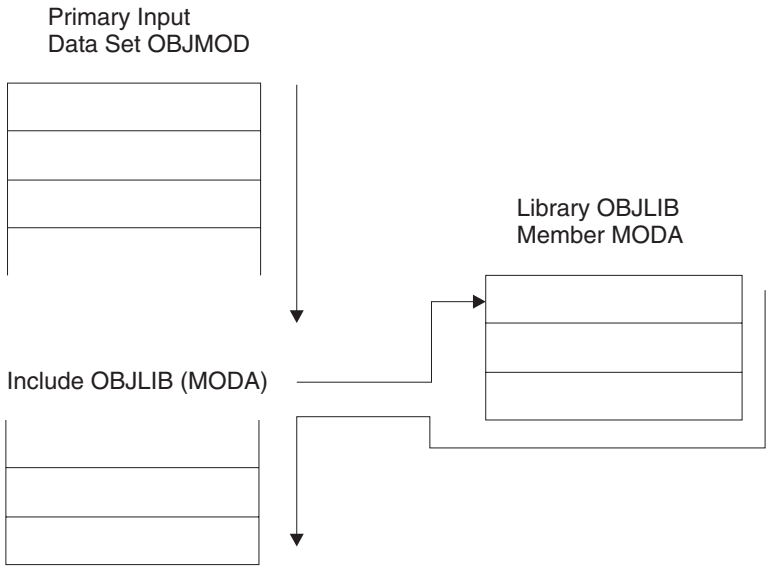


Figure 10. Processing of one INCLUDE control statement

Figure 11 demonstrates the flow of processing for nested INCLUDE statements.

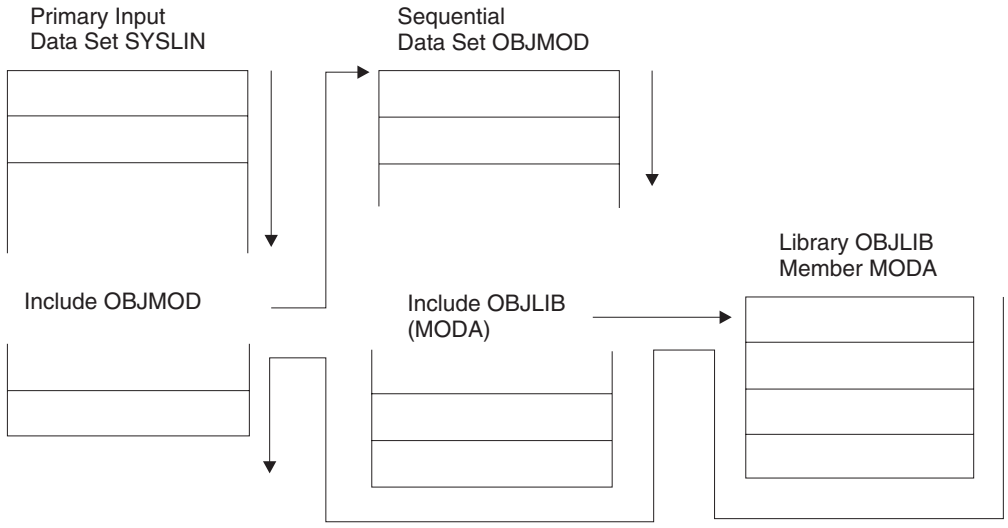


Figure 11. Processing of nested INCLUDE control statements

Including sequential data sets

Sequential data sets containing object modules or control statements, or both, can be specified by an INCLUDE control statement. In the following example, an INCLUDE statement specifies the ddnames of two sequential data sets to be used as additional input:

```
//ACCOUNTS   DD     DSNAME=PROJECT.ACCTROUT,DISP=SHR,...
//INVENTRY   DD     DSNAME=PROJECT.INVENTRY,DISP=SHR,...
//SYSLIN     DD     DSNAME=PROJECT.QTREND,...
//           DD     *
  INCLUDE    ACCOUNTS,INVENTRY
/*
```

Each ddname could have been specified on a separate INCLUDE statement. Using either method a DD statement must be specified for each ddname.

Another method of performing the preceding example is given in “Including concatenated data sets.”

Including library members

Members of a partitioned data set, PDSE or a z/OS UNIX directory, can be specified on an INCLUDE control statement. The member or file name must be specified on the INCLUDE statement and not on the DD statement describing the data set.

In the following example, one member name is specified on the INCLUDE statement.

```
//PAYROLL    DD     DSNAME=PROJECT.PAYROUTS,DISP=SHR,...
//SYSLIN     DD     DSNAME=&&CHECKS,DISP=(OLD,DELETE),...
//            DD     *
  INCLUDE    PAYROLL(FICA)
/*
```

If more than one member of a library is to be included, the INCLUDE statement specifies all the members to be used from that library. The member names appear in parentheses following the ddname of the library, and must not appear on the DD statement.

In the following example, an INCLUDE statement specifies two members from each of two libraries to be used as additional input:

```
//PAYROLL    DD     DSNAME=PROJECT.PAYROUTS,DISP=SHR,...
//ATTEND     DD     DSNAME=PROJECT.ATTROUTS,DISP=SHR,...
//SYSLIN     DD     *
  INCLUDE    PAYROLL(FICA,TAX),ATTEND(ABSENCE,OVERTIME)
/*
```

Each library could have been specified on a separate INCLUDE statement. Using either method a DD statement must be specified for each ddname.

Including concatenated data sets

Several data sets can be designated as input with one INCLUDE statement that specifies one ddname. Additional data sets are concatenated to the data set described on the specified DD statement. There are two types of concatenation, described separately below. With either type, you can concatenate data sets with unlike characteristics, such as record format and record length.

Note however, that the binder does not support concatenation of z/OS UNIX files.

Sequential concatenation

This form of concatenation is used when the INCLUDE statement provides a ddname but no member names. The concatenated data sets can be sequential files, or they can be members of partitioned data sets with the member name included in the DD statement. Each data set or member listed in the concatenation may contain a load module, a program object, or any combination of control statements and object modules.

In the following example, two sequential data sets are concatenated and then specified as input with one INCLUDE statement:

```
//CONCAT     DD     DSNAME=PROJECT.ACCTROUT,DISP=SHR,...
//           DD     DSNAME=PROJECT.INVENTRY,DISP=SHR,...
//SYSLIN     DD     DSNAME=PROJECT.SALES,DISP=OLD,...
//           DD     *
  INCLUDE    CONCAT
/*
```

When the INCLUDE statement is recognized, the contents of the sequential data sets PROJECT.ACCTROUT and PROJECT.INVENTRY are processed.

Library concatenation

This form of concatenation is used when the INCLUDE statement provides one or more member names. The concatenated data sets must all be partitioned data sets without any member name included in the DD statement. Each member referenced by the INCLUDE statement may contain a load module, a program object, or any combination of control statements and object modules.

Members from more than one library can be designated as input with one ddname on an INCLUDE statement. In this case, all the members are listed on the INCLUDE statement. The partitioned data sets or PDSEs are concatenated using the ddname from the INCLUDE statement:

```
//CONCAT     DD     DSNAME=PROJECT.PAYROUTS,DISP=SHR,...
//           DD     DSNAME=PROJECT.ATTROUTS,DISP=SHR,...
//SYSLIN     DD     DSNAME=PROJECT.REPORT,DISP=OLD,...
//           DD     *
  INCLUDE    CONCAT(FICA,TAX,ABSENCE,OVERTIME)
/*
```

When the INCLUDE statement is read, the two libraries PROJECT.PAYROUTS and PROJECT.ATTROUTS are searched for the four members and the members are processed as input. Library directories are searched in the order of library appearance in the JCL.

Resolving external references

You can request that the binder automatically search libraries to resolve external references that were not resolved during primary and secondary input processing. The binder can also process unresolved external references found in modules from additional data sources.

Note: The following discussion of automatic library call services does not apply to unresolved weak external references. They are left unresolved unless resolved to external symbols defined by modules included in the process of resolving other external references.

There are three ways to obtain automatic library call:

1. By providing AUTOCALL control statements. This is called incremental autocall and is processed at the time the control statement is encountered, using a source specified on the statement.
2. By providing LIBRARY control statements which specify sources to resolve references. Processing for these statements is deferred until all primary and secondary input sources have been exhausted.
3. By default if unresolved references remain at the end of the processing. The SYSLIB DD is used for this autocall.

There are also two ways to suppress automatic library call processing:

1. By providing an NCAL (or NOCALL) invocation option. This suppresses all automatic library call processing.
2. By providing LIBRARY control statements which specify names of external references that should not be resolved by automatic library call.

When you have requested automatic library call, the binder searches the directory of the automatic call library for an entry that matches the unresolved external reference. When a match is found, the entire member is processed as input to the binder.

Automatic library call can resolve an external reference when:

- The external reference is a member name or an alias of a module in the call library, AND
- The external reference is defined as an external name in the external symbol dictionary of a module contained in that member.

If an unresolved external reference is a member name or an alias in the library, but is not an external name in that member, the member is processed but the external reference remains unresolved unless it is subsequently defined.

When resolving external references, the binder searches the call library defined on the SYSLIB DD statement. The call library can contain program objects, load modules, or object modules and control statements (except INCLUDE, LIBRARY, and NAME).

Modules from libraries other than the SYSLIB call library can be searched by the binder as directed by the LIBRARY control statement. The library specified in the control statement is searched for member names that match specific external references that are unresolved at the end of input processing. If any unresolved references are found in the modules located by automatic library call, they are resolved by another search of the library. Any external references not specified on a LIBRARY control statement are resolved from the library defined on the SYSLIB DD statement.

To prevent the binder from automatically searching call libraries, use either the LIBRARY statement for selected unresolved external references, or the NCAL option on the EXEC statement for *all* unresolved external references. See “Directing external references to a specific library” on page 54 for a discussion of the LIBRARY control statement and the NCAL option.

Incremental autocall

The autocall phase can be invoked multiple times. Incremental autocall can be triggered at any point during primary or secondary input processing by the AUTOCALL control statement (or equivalent API call).

The library name from the autocall request will be used in the same way as SYSLIB is used in standard (final) autocall. The following functions of final autocall will not take place during incremental autocall:

- Processing of LIBRARY control statements or SETL API requests
- RES processing (see section 4.3.1)
- C Renaming logic
- Invocation of the INTFVAL exit
- Determination of Imports and Exports
- Error messages relating to unresolved references.

Autocall with C370lib data sets

The binder supports autocall from both C370lib data sets and z/OS UNIX archive libraries. A C370lib is created by the C/C++ Object Library Utility (C370LIB or EDCLIB). It is an object module library that contains a special member named @@DC370$ or @@DC390$. This special member is used as a replacement for the system directory in the autocall process to perform matches on long symbol names. In addition it preserves certain additional symbol attributes that cannot be saved in a standard MVS object library directory entry. In some cases these attributes are used by the binder to select among variant routines with matching names (see "Autocall matching for C370LIB and archive libraries.")

For each library in the SYSLIB concatenation containing the special member @@DC370$ or @@DC390$, the names in the special member take precedence over the regular directory entries for that library.

For example given a SYSLIB concatenation

```
PDSE
PDS1 (with @@DC370$ member)
PDS2
```

the actual search order would be:

```
PDSE directory names
names from @@DC370$ in PDS1
PDS1 directory names
PDS2 directory names
```

Note: @@DC370$ and @@DC390$ members are ignored during INCLUDE processing. Only member or alias names in the PDS or PDSE directory can be used to resolve member names listed on an INCLUDE statement.

Autocall with archive libraries

The binder also supports autocall from z/OS UNIX archive libraries. These archive libraries contain XOBJ or GOFF format modules and special directory information similar to that contained in C370lib object libraries.

Archive libraries are created by the UNIX System Services **ar** command. Like C370LIBs, they may contain attributes used by the binder to select among variant routines with matching names (see "Autocall matching for C370LIB and archive libraries"). Unlike C370LIBs, archives cannot be concatenated.

Note: Archive libraries cannot be used as the target for INCLUDE statements.

Autocall matching for C370LIB and archive libraries

C370LIB data sets and archive libraries contain special directory information stored by the EDCLIB procedure **ar** command respectively. Recent versions of these programs supply attribute information about the object files in the libraries, and support multiple copies of the same program in a single library with variant attribute informnation.

The binder uses some of the attribute information to choose among the variant object files. In priority order, the binder will attempt to match a called program's attributes with those declared by the caller based on:

1. 64-bit execution mode
2. Use of XPLINK linkage

3. Writable static

Searching the link pack area

When the binder is invoked for the loader function at entry IEWBLDGO, external references can be resolved to module names in the system link pack area. The link pack area is searched if the RES option is in effect. If you use the NORES option, the binder suppresses the search.

When the RES option is in effect, the library search order is:
1. Special libraries defined by the LIBRARY control statement.
2. System link pack area.
3. Automatic call libraries defined by the SYSLIB DD statement.

Dynamic symbol resolution

After final autocall processing is complete, if the DYNAM(DLL) option is in effect, the binder will attempt dynamic resolution of those symbols still unresolved. Unresolved symbols are eligible for dynamic resolution if they have a scope of import/export. Symbols on IMPORT control statements are treated as definitions, and cause a matching unresolved symbol to be considered dynamically rather than statically resolved. A dynamically resolved symbol causes an entry in the binder class B-IMPEXP to be created. The binder does not issue unresolved symbol messages for symbols that are to be dynamically resolved.

Specifying automatic call libraries

If automatic library call is requested, the call library must be a partitioned data set or PDSE described by a DD statement with a ddname of SYSLIB. Details concerning logical record lengths and record formats for SYSLIB libraries are given in “SYSLIB DD statement” on page 35. Call libraries can be concatenated.

Call libraries

Most compilers have their own automatic call libraries, which can contain input/output, data conversion, or other special routines needed to complete a module. Other products provide assembler and compiler preprocessors that generate calls to such routines in your program. You and your organization can provide additional libraries. When an object module is created, the assembler or compiler creates an external reference for these special routines. The appropriate library must be defined when an object module produced by a particular assembler or compiler is bound; the binder resolves the references from this library.

See the appropriate user’s guide for the name of the call library.

In the following example, a Fortran object module created in STEPA is bound in STEPB, and the Fortran automatic call library is used to resolve external references:

```
//STEPA      EXEC
//SYSOBJ     DD      DSNAME=&&OBJMOD,DISP=(NEW,PASS),...
   ⋮
//STEPB      EXEC
//SYSLIN     DD      DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
//SYSLIB     DD      DSNAME=SYS1.VSF2FORT,DISP=SHR
```

Concatenation of call libraries

Call libraries from various sources can be concatenated. When concatenating libraries to define input to the binder, you can combine libraries containing object modules, load modules, program objects, and control statements.

If object modules from different system processors are to be bound to form one program object or load module, the call library for each must be defined. This is accomplished by concatenating the additional call libraries to the library defined on the SYSLIB DD statement. In the following example, a Fortran object module and a COBOL object module are to be bound. The two call libraries are concatenated as follows:

```
//SYSLIB     DD      DSNAME=SYS1.VSF2FORT,DISP=SHR
//           DD      DSNAME=SYS1.COBLIB,DISP=SHR
```

Libraries typically are cataloged. No unit or volume information is needed.

Directing external references to a specific library

The LIBRARY control statement can be used to direct the binder to search a library other than that specified in the SYSLIB DD statement. This method resolves only external references listed on the LIBRARY statement. All other unresolved external references are resolved from the library in the SYSLIB DD statement.

The LIBRARY statement can also be used to specify external references that should not be resolved from the automatic call library. The LIBRARY statement specifies the duration of the unresolved condition: either during the current binder job step, called *restricted no-call*; or during this or any subsequent binder job step, called *never-call*.

Examples of each use of the LIBRARY statement follow. The syntax of the LIBRARY statement is shown in “LIBRARY statement” on page 113.

Additional call libraries

If the additional libraries are intended to resolve specific references, the LIBRARY statement must contain the ddname of a DD statement describing the library. The LIBRARY statement also contains, in parentheses, the external references to be resolved from the library; that is, the names of the members to be used from the library. If the unresolved external reference is not a member name in the specified library, no attempt is made to resolve it from SYSLIB or LPA, and the reference remains unresolved unless subsequently defined.

For example, two modules (DATE and TIME) from a system call library have been rewritten. The new modules are to be tested with the calling modules before they replace the old modules. Because the binder would otherwise search the system call library (which is needed for other modules), a LIBRARY statement is used, as follows:

```
//SYSLIB     DD     DSNAME=SYS1.COBLIB,DISP=SHR
//TESTLIB    DD     DSNAME=USER.TESTLIB,DISP=SHR,...
//SYSLIN     DD     DSNAME=PROJECT.ACCTROUT,...
//           DD     *
  LIBRARY           TESTLIB(DATE,TIME)
/*
```

Two external references, DATE and TIME, are resolved from the library described on the TESTLIB DD statement. All other unresolved external references are resolved from the library described on the SYSLIB DD statement.

Note: If a specified reference cannot be found in the designated library, it remains unresolved. No attempt will be made to resolve it from SYSLIB.

Preventing external references from being resolved

You can use the LIBRARY statement to specify those external references in the output module for which there is no library search during the current binder job step. To do this, specify the external references in parentheses without specifying a ddname. The references remain unresolved, but the binder can mark the module as executable, depending upon the value specified for the LET option.

For example, a program contains references to two large modules that are called from the automatic call library. One of the modules has been tested and corrected;

the other is tested in this job step. Rather than execute the tested module again, the restricted no-call option is used to prevent automatic library call from processing the module as follows:

```
//           EXEC   PGM=IEWBLINK,PARM=LET
//SYSLIB     DD     DSNAME=PROJECT.PVTPROG,DISP=SHR
   ⋮
//SYSLIN     DD     DSNAME=&&PAYROL,...
//           DD     *
  LIBRARY           (OVERTIME)
/*
```

As a result, the external reference to OVERTIME is not resolved.

Never-call option

You can use the never-call option to specify external references that are not to be resolved by automatic library call during this or any subsequent binder job step. To do this, put an asterisk before the external references in parentheses. The references remain unresolved but the binder marks the module as executable.

For example, a certain part of a program is never executed, but it contains an external reference to a large module (CITYTAX) which is no longer used by this program. The module is in a call library needed to resolve other references. Rather than take up storage for a module that is never used, the never-call option is specified, as follows:

```
//           EXEC   PGM=IEWBLINK,PARM=LET
//SYSLIB     DD     DSNAME=PROJECT.PVTPROG,DISP=SHR
   ⋮
//SYSLIN     DD     DSNAME=PROJECT.TAXROUT,DISP=OLD,...
//           DD     *
  LIBRARY           *(CITYTAX)
/*
```

When program TAXROUT is bound, the external reference to CITYTAX is not resolved. If the module is subsequently rebound, CITYTAX will remain unresolved unless it is bound with another module that requires CITYTAX.

NCAL option: Negating the automatic library call

When the NCAL option is specified, no automatic library call occurs to resolve external references that are unresolved after input processing. The NCAL option is similar to the restricted no-call option on the LIBRARY statement, except that the NCAL option negates automatic library call for all unresolved external references and restricted no-call negates automatic library call for selected unresolved external references. With NCAL, all external references that are unresolved after input processing is finished remain unresolved. The module is or is not marked executable depending on the value specified for the LET option.

The NCAL option is a special processing parameter that is specified on the EXEC statement as described in “CALL: Automatic library call option” on page 74.

Renaming

Binder renaming logic occurs when all possible name resolution has been performed on the original names. It allows the conversion of long mixed case names from XOBJ or GOFF object modules to short uppercase names and will redrive the autocall process. Renaming logic applies only to nonimported, renameable function references that are still unresolved and consists of the following:

1. The RENAME control statement allows users to control the renaming of specific symbols, as they could with the prelinker.
2. Standard C/C++ library functions will be renamed to the names appearing in the SCEELKED static bind library. The mappings are those defined by module EDCRNLST. If the binder is not able to locate and load this module, an informational message will be issued.
3. If UPCASE=YES is in effect, renaming will be performed approximately according to the rules used by the prelinker.

   See “UPCASE: UPCASE option” on page 93 for more information.

Chapter 5. Editing data within a program module

The binder can perform editing services either automatically or as directed by you with control statements. These editing capabilities allow you to modify programs on a section basis, so you can modify a section within a module without having to recompile the entire source program.

The editing capabilities let you modify either an entire section or external symbols within a section. Sections can be deleted, replaced, or arranged in sequence; external symbols can be deleted or changed. See “External symbols” on page 15 for an explanation of external symbols.

Any editing service is requested in reference to an *input* module. The resulting output program module reflects the request; no actual change, deletion, or replacement is made to the input module. The requested alterations are used to control binder processing, as shown in Figure 12.

Note: This chapter refers to binder processing. These concepts apply equally to linkage editor and batch loader processing unless noted otherwise in Appendix A, “Using the linkage editor and batch loader,” on page 157. The linkage editor and batch loader do not process program objects.

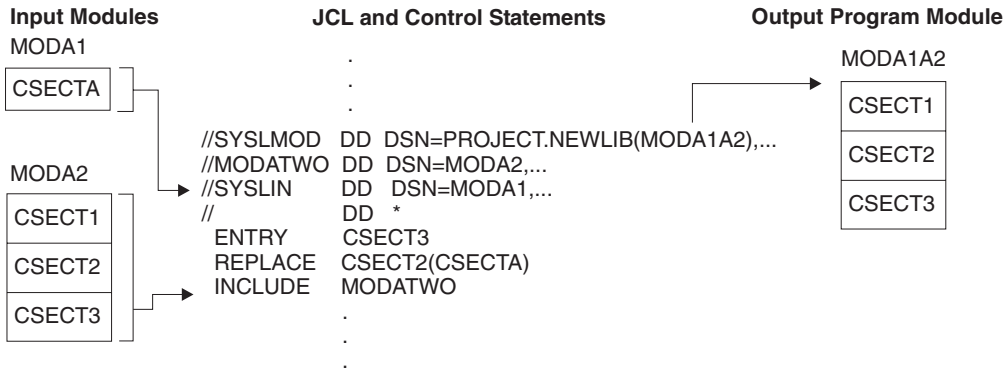


Figure 12. Editing a module. This example illustrates how the ENTRY and REPLACE statements can be used to edit a program module.

Editing conventions

When you request editing services, you should follow certain conventions to ensure that the specified modification is processed correctly.

These conventions concern the following items:
- Entry points for the new module
- Placement of control statements
- Identical old and new symbols.

Entry points

Each time the binder reprocesses a program module, the entry point for the output module should be specified in one of three ways:
- Through an ENTRY control statement
- Through an EP option when using the binder loader function

- Through an assembler- or compiler-produced END statement of an input object module if one is present. If multiple such entry point nominations are encountered, the first is used. The entry point specified on the END statement of one object module can be defined in a different object module if it is specified as an external reference in the first module.

The entry point assigned must be defined as an external name within the resulting program object or load module. The ENTRY control statement takes precedence over the EP option, which in turn takes precedence over the END statement.

Placement of control statements

The control statement (such as CHANGE or REPLACE) used to specify an editing service must immediately precede either the module to be modified or the INCLUDE statement that specifies the module. If an INCLUDE statement specifies several modules, the CHANGE or REPLACE statement applies only to the first module included.

Identical old and new symbols

The same symbol should not appear as both an old external symbol and a new external symbol in one binder run. If a section is replaced by another section with the same name, the binder handles this automatically (see “Automatic replacement” on page 60 for more information).

Changing external symbols

You can change an external symbol to a new symbol while processing an input module. External references and address constants within the module automatically refer to the new symbol. External references from other modules to a changed external symbol must be changed with separate control statements.

Both the old and the new symbols are specified on either a CHANGE control statement or a REPLACE control statement. The use of the old symbol within the module determines whether the new symbol becomes a section name, an entry name, or an external reference.

Using the CHANGE statement

The CHANGE control statement changes a section name, a common section name, an entry name, an external or weak external reference, or a pseudoregister.

The CHANGE statement must immediately precede either the input module that contains the external symbol to be changed, or the INCLUDE statement that specifies the input module. The scope of the CHANGE statement is the immediately following module.

If a CHANGE statement appears in a data set included from an automatic call library and is not immediately followed by an object module in the same data set, the request for the change is ignored.

See “CHANGE statement” on page 103 for the specific information on using the CHANGE control statement.

Example of changing external symbols

In the following example, assume that SUBONE is defined as an external reference in the input program module. A CHANGE statement is used to change the external reference to NEWMOD as shown in Figure 13.

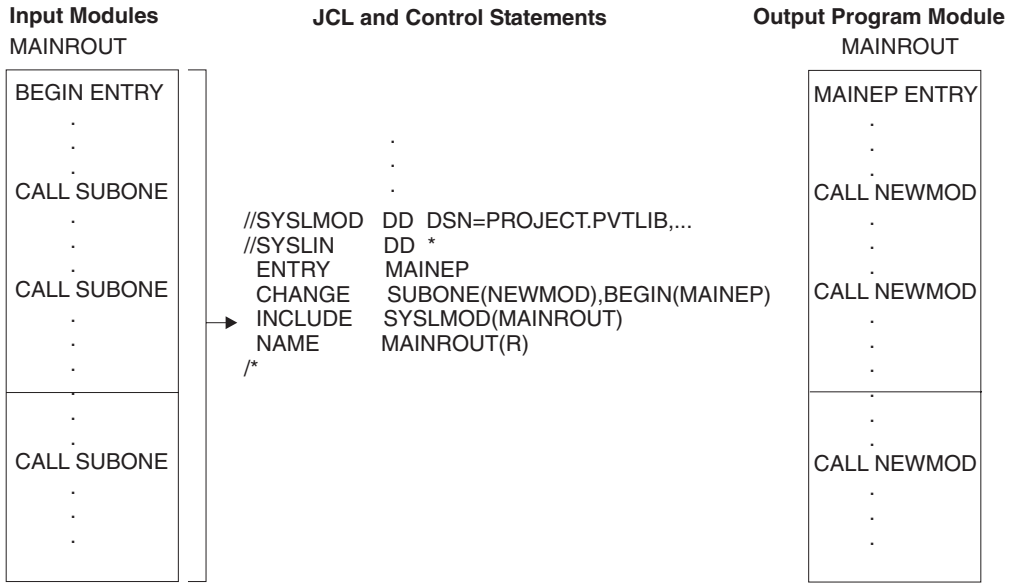


Figure 13. Changing an external reference and an entry point

In the program module MAINROUT, every reference to SUBONE is changed to NEWMOD. The INCLUDE statement specifies the ddname SYSLMOD, allowing the library to be used both as the input and the output module library.

More than one change can be specified on the same control statement. If, in the same example, the entry point is also to be changed, the two changes can be specified at once (see Figure 13).

Because the main entry point name is changed from BEGIN to MAINEP, you must use the ENTRY statement to change the library directory entry for the module to reflect the new name of the entry point.

Replacing sections

An entire section can be replaced with a new section. Sections can be replaced either automatically or with a REPLACE control statement. Automatic replacement acts upon all input modules; the REPLACE statement acts only upon the module that follows it.

Notes:

1. Any CSECT identification records (IDR) associated with a particular section are also replaced.
2. **For assembler language programmers only**: When some but not all sections of a separately assembled module are to be replaced, the binder causes A-type address constants that refer to a deleted symbol to be incorrectly resolved unless the entry name is at the same displacement from the origin in both the old and the new section. If all sections of a separately assembled module are replaced, no restrictions apply.

Automatic replacement

Sections are automatically replaced if both the old and the new section have the same name. The first of the identically named sections processed by the binder is made a part of the output module. All subsequent sections with that name are ignored; external references to identically named sections are resolved with respect to the first one processed. Therefore, to cause automatic replacement, the new section must have the same name as the section to be replaced, and must be processed before the old section.

Attention: Automatic replacement applies to duplicate section names only. If duplicate entry points exist in sections with different names, a REPLACE control statement must be used to specify the entry point name.

Example 1: Object module with two sections

An object module contains two sections, READ and WRITE; member INOUT of library PROJECT.PVTLIB also contains a section WRITE.

```
//SYSLMOD    DD     DSNAME=PROJECT.PVTLIB,DISP=OLD
//SYSLIN     DD     *

Object Deck for READ
Object Deck for WRITE

  ENTRY      READIN
  INCLUDE    SYSLMOD(INOUT)
  NAME       INOUT(R)
/*
```

The output module contains the new READ section, the replacement WRITE section, and all remaining sections from INOUT.

Example 2: Large program module with many sections

A large module named PAYROLL, originally written in COBOL, contains many sections. Two sections, FICA and STATETAX, were recompiled and passed to the binder job step in the &&OBJECT data set. Then, by including the &&OBJECT data set before the program module PAYROLL (a member of the program library PROJECT.LIB001), the modified sections automatically replace the identically named sections. See Figure 14 on page 61.

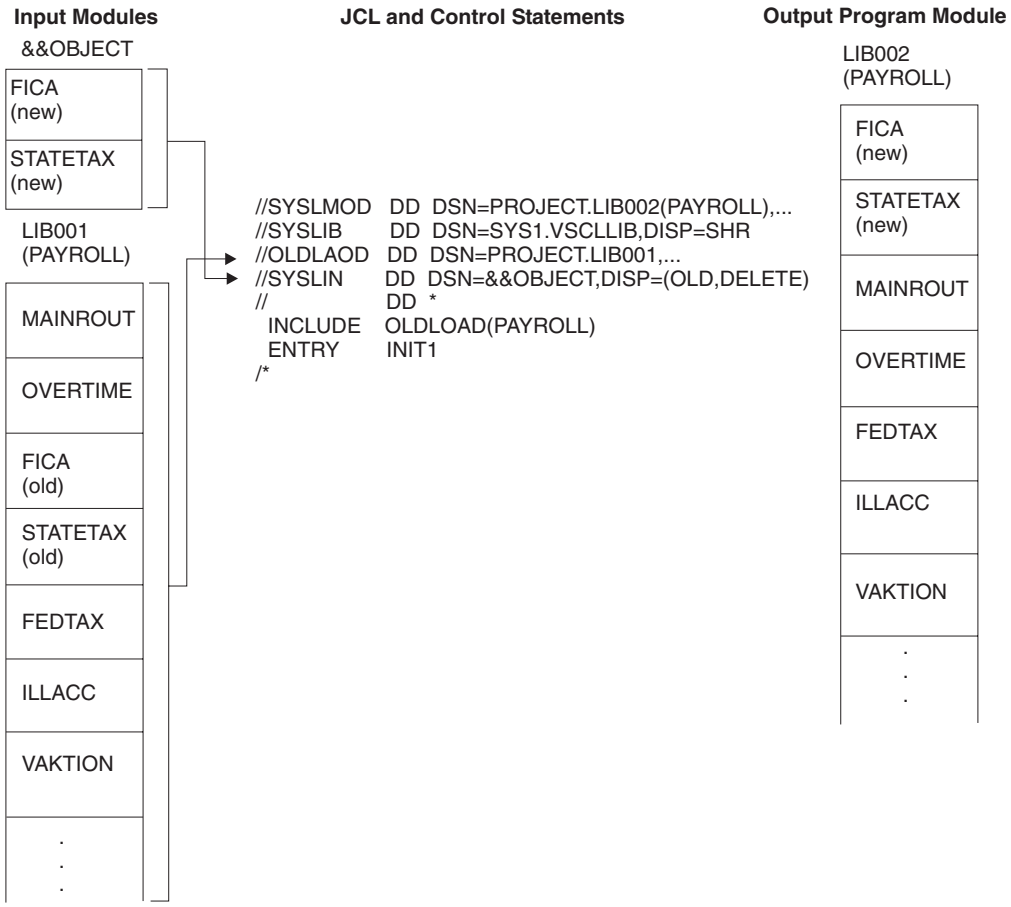


Figure 14. Automatic replacement of sections

The output module contains the modified FICA and STATETAX sections and the rest of the sections from the old PAYROLL module. The main entry point is INIT1, and the output module is placed in a library named PROJECT.LIB002. The COBOL automatic call library is used to resolve any external references that might be unresolved after the SYSLIN data sets are processed. The new module is named PAYROLL because PAYROLL is specified as the member name on the SYSLMOD DD statement and was not overidden by a NAME control statement.

Using the REPLACE statement to replace sections and named common areas

The REPLACE statement is used to replace sections and named common areas (also called *common sections*) by providing old and new section names. The name of the old section appears first, followed by the name of the new section in parentheses.

The scope of the REPLACE statement is the immediately following module. The REPLACE statement must precede either the input module that contains the section to be replaced, or the INCLUDE statement that specifies the input module. The replacing section can be either before or after the replaced section in the binder input. If a REPLACE statement appears in a data set included from an automatic call library and is not immediately followed by an object module in the same data set, the request is ignored.

An external reference to the old section (or area) from within the same input module is resolved to the new section. An external reference to the old section from any other module becomes an unresolved external reference unless one of the following occurs:

- The external reference to the old section is changed to the new section with a separate CHANGE control statement.
- The same entry name appears in the new section or in some other section in the binder input.

In the following example, the REPLACE statement is used to replace one section with another of a different name. Assume that the old section SEARCH is in library member TBLESRCH, and that the new section BINSRCH is in the data set &&OBJECT, which was passed from a previous step as shown in Figure 15.

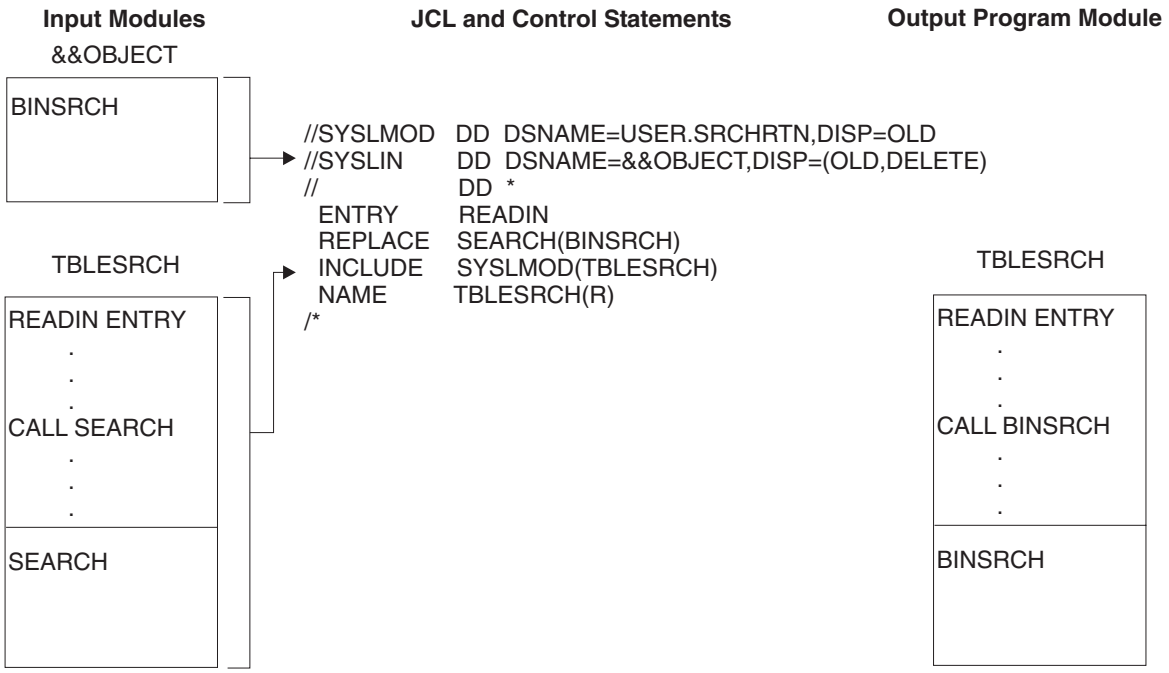


Figure 15. Replacing a section with the REPLACE control statement

The output module contains BINSRCH instead of SEARCH; any references to SEARCH within the module refer to BINSRCH. Any external references to SEARCH from other modules will not be resolved to BINSRCH.

See “REPLACE statement” on page 122 for more information on using the REPLACE statement.

Deleting external symbols

The REPLACE statement can be used to delete an external symbol. The external symbol can be a named section, a named common area, an entry point, a strong or weak external reference, or a pseudoregister. The REPLACE statement must immediately precede either the module in the input data set that contains the external symbol to be deleted or the INCLUDE statement in the job stream that

specifies the module. Only one symbol appears on the REPLACE statement; the appropriate deletion is made depending on how the symbol is defined in the module.

If the symbol is a section name, the entire section is deleted. The section name is deleted from the external symbol dictionary only if no address constants refer to the name from within the same input module. If an address constant does refer to it, the section name is changed to an external reference. Any CSECT identification data associated with that section is also deleted.

The preceding is also true of an entry name to be deleted. Any references to it from within the input module cause the entry name to be changed to an external reference.

For external references and pseudoregisters, the symbol is deleted only if no RLD contains references to the ESD entry to be deleted.

These editor-supplied external references, unless resolved with other input modules, cause the binder to attempt to resolve them from the automatic call library. Also, the deletion of an external symbol in an input module might cause external references from other input modules to be unresolved. Either condition can cause the output module to be marked not executable.

If you delete a section that contains any unresolved external references, those references are removed from the external symbol dictionary.

In the example shown in Figure 16, the section CODER is deleted. If no address constants refer to CODER from other sections in the module, the section name is also deleted. If address constants refer to CODER, the name is retained as an external reference.

See “REPLACE statement” on page 122 for more information on using the REPLACE statement.

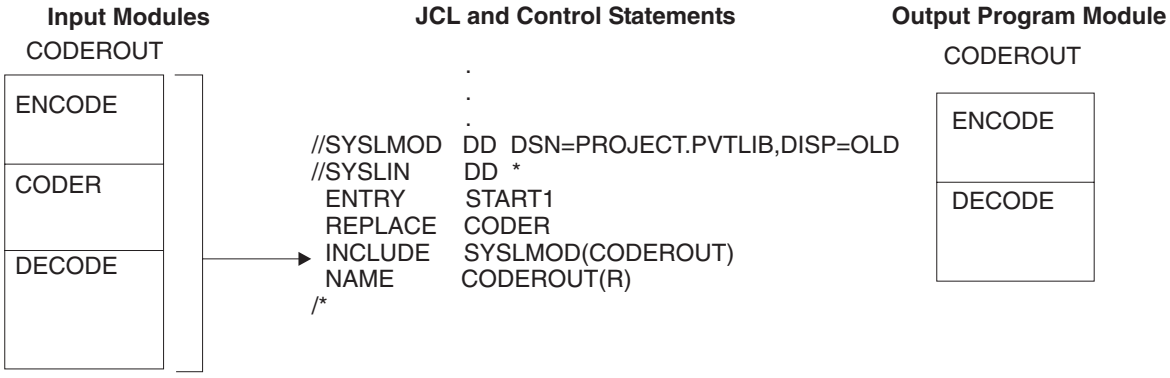


Figure 16. Deleting a section

Ordering sections or named common areas

The sequence of sections or named common areas in an output module can be specified by using the ORDER control statement.

Normally, the order that sections are received during input processing are preserved in the resulting module. Common areas are placed at the end. You can change the section order by coding one or more ORDER control statements.

Individual sections or named common areas are arranged in the output module according to the sequence in which they appear on the ORDER control statement. Multiple ORDER control statements can be used in a job step. The sequence of the ORDER statements determines the sequence of the sections or named common areas in the load module or program object.

Any sections or named common areas that are not specified on ORDER statements appear last in the output load module in their original sequence. If a section or named common area is changed by a CHANGE or REPLACE control statement, the new name must be used on the ORDER statement.

In the following example, ORDER statements are used to specify the sequence of five of the six sections in an output module. A REPLACE statement is used to replace the old section, SESECTA, with the new section, CSECTA, from the data set &&OBJECT, which was passed from a previous step. Assume that the sections to be ordered are found in library member MAINROOT shown in Figure 17.

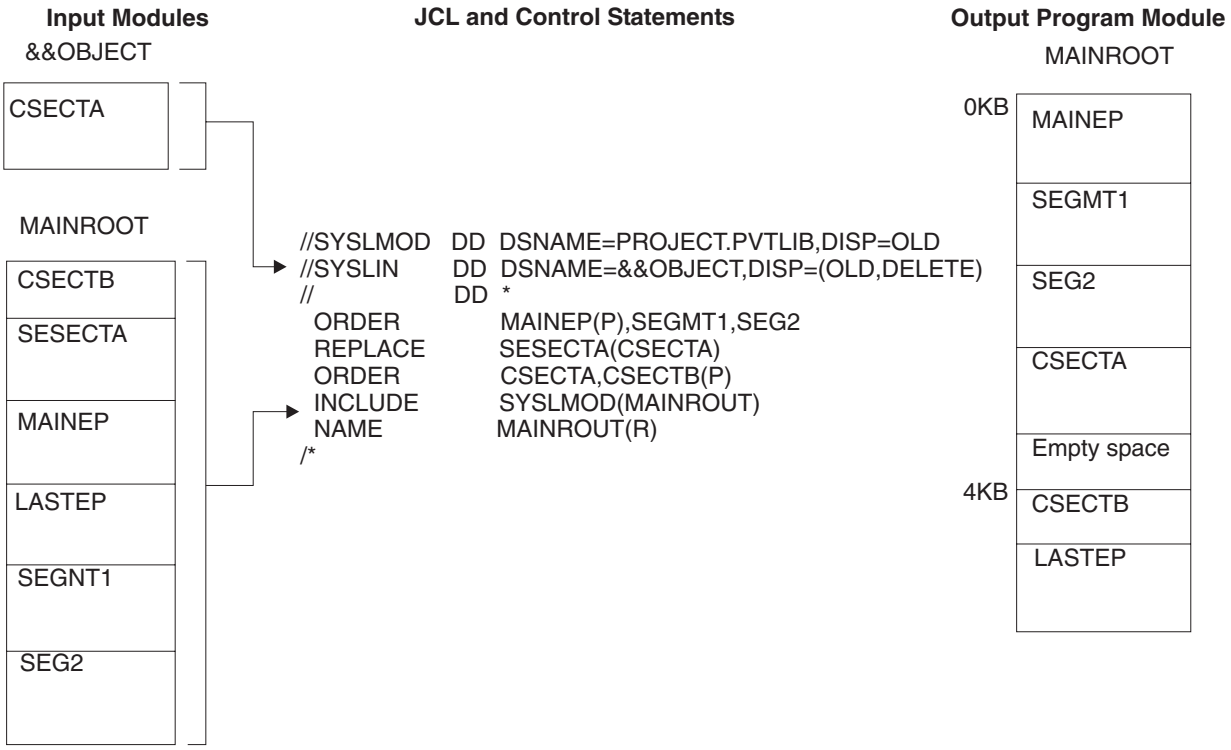


Figure 17. Ordering sections

In the load module MAINROOT, the sections MAINEP, SEGMT1, SEG2, CSECTA, and CSECTB are rearranged in the output load module according to the sequence specified in the ORDER statements. A REPLACE statement is used to replace section SESECTA with section CSECTA from data set &&OBJECT, which was passed from a previous step. The ORDER statement refers to the new section

CSECTA. Section LASTEP appears after the other sections in the output module, because it was not included in the ORDER statement operands. The order control statement cannot be used to order parts.

Note that empty space is inserted in the module before CSECTB. This is done to ensure page alignment for CSECTB as specified by the "(P)" operand on the ORDER control statement (this is discussed in "Aligning sections or named common areas on page boundaries").

See "ORDER statement" on page 117 for specific information on using the ORDER statement.

Aligning sections or named common areas on page boundaries

You can use either the ORDER statement or the PAGE statement to place a section or named common area on a page boundary. This allows you to operate with a lower paging rate, making more efficient use of real storage.

The section or common area to be aligned is named on either the PAGE statement or the ORDER statement with the P operand. If any sections in the module are to be page aligned the module is loaded on a page boundary. For multitext class program objects, a page-align request for a section will cause each text element within the section to be aligned on a page boundary.

In the following example, the sections RAREUSE and MAINRT are aligned on page boundaries by PAGE and ORDER control statements. Sections MAINRT, CSECTA, and SESECT1 are sequenced by the ORDER control statement. Assume that each section is 3KB in length as shown in Figure 18 on page 66.

The binder places the sections MAINRT and RAREUSE on page boundaries. Sections MAINRT, CSECTA, and SESECT1 are sequenced as specified in the ORDER statement. RAREUSE, while placed on a page boundary, appears after the sections specified in the ORDER statement because it was not specified on the ORDER statement.

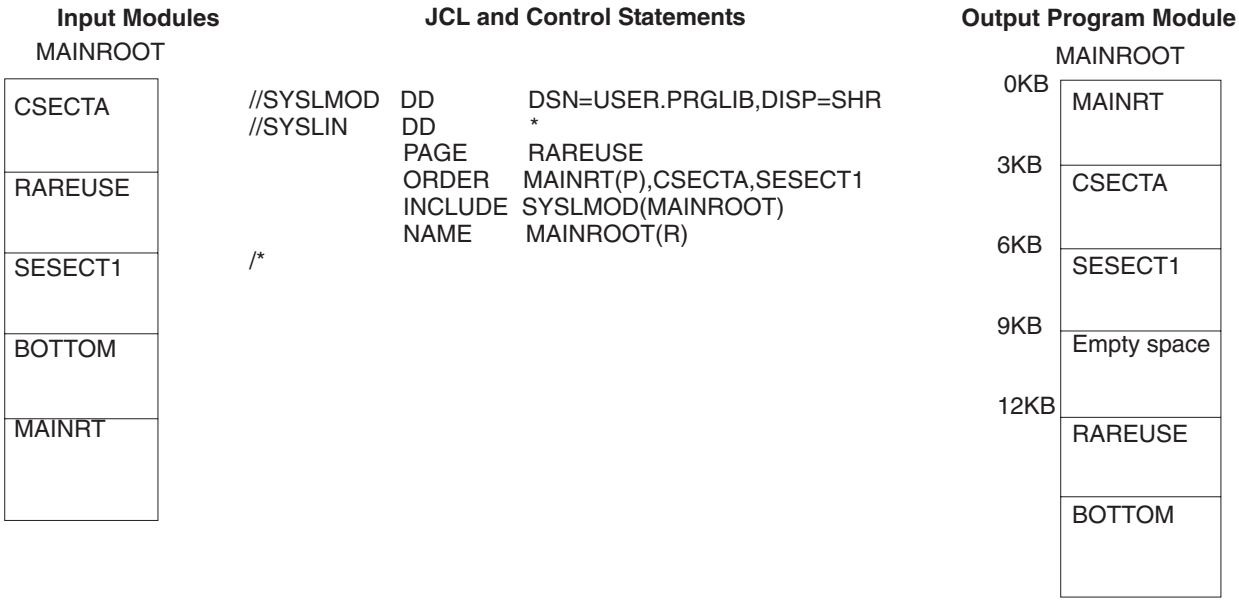


Figure 18. Aligning sections on page boundaries

For more information on using these control statements, see “ORDER statement” on page 117 and “PAGE statement” on page 120.

Chapter 6. Binder options reference

This section describes the processing and attribute options that can be requested. Options for a particular binder execution can be provided on the PARM field of the EXEC statement. Options also can be written to a data set and included using the OPTIONS keyword in the PARM field.

If more than one output module is produced by the same binder job step, the options specified will apply to each output module. Some options define attributes of the output modules. The attributes for each module are stored in the directory of the program library along with the member name.

Note: This chapter refers to binder processing. These concepts apply equally to linkage editor and batch loader processing unless noted otherwise in "Processing and attribute options reference" on page 162. The linkage editor and batch loader cannot process program objects.

The following binder options allow you to override the defaults set for your installation.

Specifying binder options

Binder options are usually specified in the PARM field of the EXEC statement. The syntax of the PARM field is:

PARM=(*option[,option],...*)

where option can be specified as

{{*option*}
{*option(value[,value]...)*}
{*option=value*}
{*option=(value[,value]...)*}}

You can use single quotations marks, rather than parentheses, to enclose the complete options string in the PARM field. You can use parentheses outside a complete string that is delimited by single quotation marks, as in PARM=('option,option'). You cannot use single quotation marks outside a complete string that is delimited by parentheses. You can enclose values in parentheses.

Binder keywords are always converted to upper case. You must use commas between options. You cannot use blanks within the options string. If you only specify one option, it need not be enclosed in parentheses or single quotation marks.

The binder bridges the limitations imposed by the JCL interpreter by allowing additional freedom in the format of the options string. While it makes every effort to resolve explicit (and implied) syntactical and semantic combinations in the options string, its success is very much dependent on the validity of the string specification. Caution and adherence to the options syntax is recommended when building the options string. Binder warning or error messages will identify any problems detected while parsing the options string.

Options that would otherwise be set on the PARM field can also be specified in the *options file*. This allows you to specify a set of binder options that might otherwise exceed the MVS PARM string length limitation of 100 bytes. It also allows you to create one or more binding profiles that can be included at bind time. Options are processed in order, starting with the beginning of the parm string. When you specify OPTIONS=ddname in the PARM field, the ddname is opened and the options in that file are processed. Processing then continues with the option following OPTIONS= in the parm string.

Many options have as possible values *YES* and *NO*. These options usually have an associated option that begins with *N* or *NO*. For example, you can specify **MAP** to produce a module map and **NOMAP** to suppress production of a module map. You can also specify the MAP option as **MAP**=*YES* or **MAP(***YES***)** and **MAP**=*NO* or **MAP(***NO***)**. The summary of binder options in Table 6 on page 69 shows the associated negative option if the options values are *YES* and *NO*.

The options you specify in the PARM parameter of the EXEC statement *always* override similar data from included modules. For example, if you specify PARM=RENT, the resultant module is marked “reentrant” regardless of the reusability of any included modules.

Establishing installation defaults

Note: This procedure involves making changes to an MVS component and modifying an authorized library. It can only be performed by your system programmer, who can provide you with a list of the defaults that have been established for your installation.

Default values for binder options can be tailored to the needs of your installation by replacing a data-only load module in SYS1.LINKLIB. CSECT IEWBODEF in load module IEWBLINK can be used to specify default values for one or more options. The module is initially set to the null string but can be modified to contain binder option default settings in the same form they would be specified in the JCL PARM field, without enclosing apostrophes or parentheses. A halfword length field preceding the string must be set to the current length of the string.

The following example shows how you can set the defaults for binder options FETCHOPT, LET and MSGLEVEL:

```
IEWBODEF  CSECT
          DC     AL2(ENDPARM-PARMS)
PARMS     EQU    *
          DC     C'FETCHOPT(NOPACK,NOPRIME),'
          DC     C'LET(4),'
          DC     C'MSGLEVEL(4)'
ENDPARM   EQU    *
          END
```

Any options that can be set in the JCL PARM field can be defaulted in this manner. Notice the setting of the length and the comma delimiters between options. Once this module has been assembled it must be linked into module IEWBLINK, using either the linkage editor or binder, replacing the empty module.

It is recommended that only binder processing options, (for example, COMPAT, LINECT, LIST, MAP, and MSGLEVEL), be established for your installation. Module attributes, such as RMODE, REUS and OVLY, tend to vary from module to module, and changing the defaults for these attributes can result in unwanted conflicts. Note

that utilities, such as IEBCOPY, invoke the binder to perform part of their processing, and any defaults established with this procedure can affect those utilities as well.

Binder options

Table 6 briefly describes all of the PARM options available to the binder. For options with only yes and no values, the binder provides negative options. You can either specify the negative option or set the primary option equal to *NO*. These options are listed in parentheses beneath the primary option. Descriptions are for the primary options. Table 6 also lists the default values for each option when using either IEWBLINK or IEWBLDGO.

Most options can be set on the PARM field of the EXEC statement or on the SETOPT control statement. Options set from the PARM field are in effect for the entire job step, whereas options set via control statements (MODE, SETCODE, SETOPT, SETSSI) are in effect only for the module in process. Options set on control statements override settings from the PARM field.

Certain options are designated as “environmental” options and can only be specified on the PARM field (they cannot be specified in the options file). Environmental options are:

- COMPAT–Specify binder compatibility level
- LINECT–Specify number of lines per printed page
- MSGLEVEL–Specify minimum severity level for message printing
- OPTIONS–Specify embedded options file
- PRINT–Direct messages to SYSLOUT data set
- SIZE–Restrict binder working storage limitations
- TERM–Create copy of messages in SYSTERM
- TRAP–Specify error recovery environment

Detailed information on each option follows this summary.

Table 6. Summary of processing and attribute options

Option	Default values	Description
AC	0	Assigns an authorization code to the output module, which determines whether the module can use restricted system services.
ALIASES	NO	ALIASES(ALL) allows you to mark external symbols as aliases when binding a module. The resultant aliases are nonexecutable. They are simply used for symbol resolution.
ALIGN2 (NOALIGN2)	NO	Specifies that a page specification causes the text to be aligned on a 2 KB boundary within the module. It has no effect on where the module is loaded in virtual storage.
AMODE	Default is the ESD AMODE value.	Assigns an addressing mode (24, 31, 64, or ANY) to the entry points in the output program module. Specifying MIN causes the AMODE to be set to the most restrictive AMODE value of all control sections within the module.
CALL (NCAL, NOCALL)	YES	Causes the binder to search program libraries to resolve external references (automatic library call).

Table 6. Summary of processing and attribute options (continued)

Option	Default values	Description
CASE	UPPER	Controls case sensitivity in names encountered in modules, control ststements and options.
COMPAT	The earliest program object format which will support the features contained in the module.	Specifies the binder compatibility level.
DC (NODC)	NO	Causes a maximum record size of 1024 bytes to be used for the output module. (This option is only valid when creating load modules.)
DCBS (NODCBS)	NO	Allows you to specify the block size for the SYSLMOD data set in the DCB parameter of the SYSLMOD DD statement. (This option is only valid when creating load modules.)
DYNAM	NO	Determines whether the resultant module is enabled for dynamic binding. If enabled, the module becomes a DLL module from which other DLLs' imports can be resolved. Similarly, it is also able to import symbols from other DLLs.
EDIT (NE)	YES	Saves modules in a format that allows them to be rebound.
EP	no default	Specifies the external name to be used as the entry point of the loaded program.
EXITS	no default	Specifies (one or more) exits are to be taken during binder processing.
EXTATTR	no default	Specifies extended attributes for SYSLMOD when saved in a z/OS UNIX file.
FETCHOPT	NOPACK NOPRIME	Specifies how a program object should be paged-mapped (loaded) into virtual storage for execution.
FILL	no default	Specifies the character to be used to fill uninitialized areas. FILL applies to program objects only.
GID	no default	Specifies the group ID attribute to be set for the SYSLMOD file.
HOBSET	NO	Specifies if the high order bit of each V-con is to be set according to the AMODE of the target entry point.
LET (NOLET)	4	Specifies a severity code; the output module is marked as not executable if a severity code higher than the level you specified is found during processing.
LINECT	60	Specifies the number of lines to be included on each page of binder output listings. The minimum supported value is 24.
LIST (NOLIST)	OFF	Controls the information included in the SYSPRINT or SYSLOUT data set.

Table 6. Summary of processing and attribute options (continued)

Option	Default values	Description
MAP (NOMAP)	NO	Produces a module map.
MAXBLK	no default	Specifies the maximum size of a text record in a load module. This can avoid reblocking when copying to a different device type at a later time. (This option is only valid when creating load modules.)
MSGLEVEL	0	Limits the messages displayed to a given severity level and higher.
NAME	**GO	Specifies a name to be used to identify the loaded program to the system.
OL (NOOL)	NO	Brings the module into virtual storage only by using a LOAD macro.
OPTIONS	no default	Embeds a data set containing binder options to be used during the current processing.
OVLY (NOOVLY)	NO	Places the output program module in an overlay structure.
PATHMODE	Default allows file owner permission for read, write, and execute	Specifies pathmode to be used when saving a module to a z/OS UNIX file.
PRINT (NOPRINT)	YES	Indicates that informational and diagnostic messages are to be written to the SYSLOUT data set for IEWBLDGO and SYSPRINT data set for IEWBLINK.
RES (NORES)	IEWBLDGO=YES IEWBLINK=NO	Specifies whether or not the binder should automatically search the link pack area queue during automatic library call. For IEWBLDGO the default is YES, and for IEWBLINK the default is NO.
REUS	NONE	Specifies whether the output program module will be refreshable, reenterable, serially reusable or nonreusable.
RMODE	Default is the ESD RMODE value.	Assigns the residence mode (24, ANY, or SPLIT) to the output program module.
SCTR	NO	Builds control blocks needed by the system nucleus. Load module only.
SIZE	no default	Specifies the amount of virtual storage available for binder processing and the output module buffer. We do not recommend use of this option with the binder.
SSI	no default	Specifies hexadecimal information to be placed in the system status index; also see “SETSSI statement” on page 125.
STORENX (NOSTORENX)	NO	Allows the binder to replace an executable copy of a program module with a nonexecutable copy.
TERM (NOTERM)	NO	Copies the numbered binder error and warning messages into a data set that has been defined by a SYSTERM DD statement.

Table 6. Summary of processing and attribute options (continued)

Option	Default values	Description
TEST (NOTEST)	NO	Specifies that the module is to contain symbol tables in the format supported by TSO TEST.
TRAP	ON	Controls the extent of error recovery from program checks and abends, and the techniques the binder uses for it. The suboptions that can be specified are ON, OFF and ABEND.
UID	no default	Specifies a user ID attribute to be set for the SYSLMOD file.
UPCASE	NO	Indicates whether additional renaming is done when symbols remain unresolved after the binder's autocall process.
WKSPACE	See "WKSPACE: Working space specification option" on page 93.	Specifies the maximum amount of virtual storage available for binder processing both above and below the 16 MB line.
XCAL (NOXCAL)	NO	Controls whether valid exclusive references between overlay segments should be treated as a warning (severity 4) or error (severity 8) condition.
XREF (NOXREF)	NO	Produces a cross-reference table of the output module in the diagnostic output data set.

AC: Authorization code option

An output program module is assigned an authorized program facility (APF) authorization code that determines whether the module can use restricted system services and resources. You can assign an authorization code on the PARM field by using the AC parameter as follows:

AC=*n*

The authorization code *n* must be an integer between 0 and 255. The AC option default is 0. The authorization code assigned in the PARM field is overridden by an authorization code assigned through the SETCODE control statement.

A nonzero authorization code has an effect only if the program resides in an APF-authorized library defined by your system programmer. See *z/OS MVS Programming: Authorized Assembler Services Guide* for more information on APF and system integrity.

ALIASES: ALIASES option

The ALIASES option requests directory entries be created for defined symbols in a module so that those names can be used to resolve references during autocall. Because the aliases are only used for symbol resolution and are not executable, they are called "hidden" aliases. You can code the ALIASES option in the PARM field as follows:

```
ALIASES={NO | ALL}
```

Notes:

1. Hidden aliases will not be created if NO is specified, or if the ALIASES option value is defaulted. Note that the creation of hidden aliases is also dependent on the processing level of the binder. Be sure that the COMPAT processing option is at least PM3 for the ALIASES option to take effect.
2. This processing option is intended to enable standard system support for symbol resolution similar to that provided by C370LIB object libraries.
3. The DESERV macro has a new HIDE parameter that can be used by an application program to control whether or not hidden aliases are returned on a GET_ALL request. See the DESERV macro in *z/OS DFSMS Macro Instructions for Data Sets*

ALIGN2: 2KB page alignment option

When binder page-aligns sections of text, a 4KB page size is assumed. For compatibility with older environments that used 2KB pages, if you are binding program modules that will execute on hardware that supports 2KB pages (not System/370 or System/390), you can request 2KB page alignment by coding the ALIGN2 option in the PARM field of the EXEC statement. There are advantages to using 2KB alignment for modules that are executed on System/370 or System/390, although the system loader loads the module on a 4KB page boundary regardless of the ALIGN2 specification. Program data areas that are aligned are easier to read in a SNAP or ABEND dump and performance-critical assembler routines might perform better if they are aligned on 32-or 64-byte boundaries. ALIGN2 can give a smaller module without sacrificing these advantages.

```
{ALIGN2 | ALIGN2=NO | NOALIGN2}
```

ALIGN2=NO is the default value and can be specified with the keyword NOALIGN2.

AMODE: Addressing mode option

To assign the addressing mode for all the entry points into a program module (the main entry point, its true aliases, and all the alternate entry points), you should code the AMODE parameter as follows:

```
AMODE={24 | 31 | 64 | ANY | MIN}
```

The addressing mode must be either 24, 31, 64, ANY, or MIN. When AMODE=MIN is coded, the binder assigns one of the other four values to the output module; it selects the most restrictive mode of all control sections within the output module. See “Addressing and residence modes” on page 26 for more information about AMODE and RMODE.

The addressing mode assigned in the PARM field is overridden by an addressing mode assigned in the MODE control statement. However, the values in the PARM field override the separate addressing modes found in the ESD data for the control sections or private code where the entry points are located.

AMODE and RMODE values are specified independently, but the values are checked for conflicts before output processing occurs. See “AMODE and RMODE combinations” on page 27 for information on AMODE and RMODE compatibility and the setting of default values.

The AMODE keyword can also be specified as AMOD.

CALL: Automatic library call option

During processing, the binder automatically searches the library defined in the SYSLIB DD statement to resolve external references. Through AUTOCALL control statements, the binder can also be prompted to resolve external references against a specific library as said references emerge during dynamic binding.

You can turn this feature off by coding the option NOCALL or NCAL in the PARM field as follows:

```
{NCAL | NOCALL}
```

When the no automatic library include option is specified, the binder does not search any library members to resolve external references. Unresolved external references will be treated as severity 4 errors. If this option is specified, you do not need to use the LIBRARY statement to negate the automatic library call for selected external references, and you do not need to supply a SYSLIB DD statement.

Unless the LET option is also specified, other errors might still cause the module to be marked not executable.

Note: If autocall processing is disabled, references to modules in the C run-time library will not be resolved.

CASE: Case control option

You can control the binder’s sensitivity to case by coding the CASE option as follows:

```
CASE={UPPER | MIXED}
```

The case can be either **UPPER** or **MIXED**. When **CASE=MIXED** is specified,

- The binder distinguishes between upper and lower case letters, treating two strings as different if their cases do not match exactly.
- The binder does not convert any lowercase letters in names encountered in input modules, control statements, and binder options.

Binder keywords are always converted to upper case.

CASE=UPPER is the default value, causing conversion of all lower case letters to upper case during binder processing.

COMPAT: Binder level option

The COMPAT option allows you to specify the compatibility level of the binder. For instance, when binding a module you can specify LKED which will partially alter the

binder's behavior and its ultimate output as if you had invoked the linkage editor. PM2 or PM3 would allow you to take advantage of the functions supported by the newer version of program modules.

Awareness of the function provided by each option value allows you to anticipate the behavior of your bound programs as you share them across systems that might not support the same functionality. The functional differences are broadly discussed below for each option value.

If the output is directed to a PDS, the output module is saved as a load module regardless of the value of COMPAT. COMPAT(LKED) will alter some of the processing.

If SYSLMOD is allocated to a PDSE or a z/OS UNIX file, the output is saved as a program object in the format specified by the COMPAT option. If the user specified a COMPAT value that does not support the contents of the module, binder will issue a level 12 message and fail the bind.

```
COMPAT={MIN   | LKED | {CURRENT | CURR} | PM1 | PM2
          | {PM3 | OSV2R8 | OSV2R9 | OSV2R10 | ZOSV1R1 | ZOSV1R2}
          | {PM4 | ZOSV1R3 | ZOSV1R4} | {ZOSV1R5 | ZOSV1R6}}
```

CURRENT or CURR
Specifies that the output is to be as defined for the current level of the binder. For the level of Program Management support described in this version of the manual, CURRENT is the same as ZOSV1R5.

ZOSV1R5 | ZOSVIR6
COMPAT=ZOSV1R5 is the minimum level that can be specified if RMODE 64 has been specified by a compiler for deferred load data segments.

PM4 | ZOSV1R3 | ZOSV1R4
COMPAT=PM4 is the minimum level that can be specified if any of the following features are used:

- Input modules contain 8-byte adcons
- Any ESD record is AMODE 64
- Input contains symbol names longer than 1024, unless EDIT=NO
- A value of 64 is specified on the AMODE option or control statement

If COMPAT=PM4 and OVLY are both specified, COMPAT=PM4 is changed to PM1. PM4 supports all PM3, PM2 and PM1 features.

PM3 | OSV2R8 | OSV2R9 | OSV2R10 | ZOSV1R1 | ZOSV1R2
In general, COMPAT=PM3 is the minimum level that can be specified if any of the following features are used:

- Binding modules compiled using the XPLINK attribute
- DYNAM=DLL
- XOBJ format input to the binder without going through the Language Environment prelinker, or rebinding modules containing input from such sources
- Hidden aliases (from ALIASES control statement)
- Support for deferred classes or intialized text in merge classes in GOFF format input modules or data buffers passed via the binder API.

If COMPAT=PM3 and OVLY are both specified, COMPAT=PM3 is changed to PM1.

PM3 supports all PM2 and PM1 features.

PM2
In general, COMPAT=PM2 is the minimum level that can be specified if any of the following are used:

- User-defined classes passed in GOFF format input as well as certain other information supported only in GOFF format
- Names (from input modules or created by control statements which cause renaming) that are longer than 8 bytes.
- Use of RMODE=SPLIT

If OVLY is specified, COMPAT=PM2 is changed to PM1.

PM2 supports all PM1 features.

PM1
This is the minimum level which supports binder program objects. In addition to old linkage editor load module features, program object features supported here include:

- Device-independent record format
- Text length greater than 16 megabytes
- More than 32,767 external names

OVLY is supported, and will force PM1 to be used.

MIN
This is the default, and indicates that the binder should select the minimum PM level that supports the features actually in use for the current bind.

LKED
Specifies that certain binder processing options are to work in a manner compatible with the linkage editor. Specific processing affected by this specification includes:

- AMODE/RMODE—Where conflicts exist between the AMODE or RMODE of individual entry points or sections and the value specified in the AMODE or RMODE option, the option specification will prevail. No warning message will be issued and the return code remains unchanged.
- REUS—If a section is encountered in a module with a lower reusability than that specified on the REUS option, the reusability of the module is automatically downgraded. An information message is issued and the return code remains unchanged.

This should *not* be thought of as a level below PM1.

If COMPAT is not specified, the output format used by the binder will be the same as if you had specified COMPAT=MIN.

DC: Downward compatible option

If you have a need to restrict the program library block size to 1024 bytes you can specify that a maximum record size of 1024 bytes be used for the program library.

Specify the downward compatible attribute by coding **DC** in the PARM field.

{DC | DC=NO | NODC}

DC affects only load module contents, not program objects.

Specifying the DC attribute sets the block size for the program library data set to 1024 bytes with the following exception. For an existing data set, if the current block size is greater than 1024 bytes, the load module is written using a maximum record size of 1024 bytes; the block size in the DSCB entry for the data set is not changed.

DC=NO is the default value and can also be specified with the keyword **NODC**.

DCBS option

The DCBS option allows you to specify the block size for the SYSLMOD data set in the DCB parameter of the SYSLMOD DD statement. If the DCBS option is specified, the existing block size for the SYSLMOD data set can be overridden.

{DCBS | DCBS=NO | NODCBS}

If the DCBS option is specified, but no block size value is provided in the SYSLMOD DD statement, the binder uses the maximum record size for the device. If the DCBS option is not specified, but a block size value is provided in the DCB parameter of the SYSLMOD DD statement, the block size value is ignored.

The minimum block size for the SYSLMOD data set is 256 bytes. For an existing data set, the minimum block size must be less than the block size in the DSCB.

The specified block size is used unless it exceeds the maximum record size for the device or it is less than the minimum block size. In those cases, the maximum record size or minimum block size is substituted, respectively. If DCBS is specified, each CSECT starts a new block.

The following example shows the use of the DCBS option for an IBM 3380 Direct Access Storage device:

```
//LKED       EXEC   PGM=IEWBLINK,PARM='XREF,DCBS'
//SYSLMOD    DD     DSNAME=PROJECT.LOADMOD(TEST),DISP=(NEW,CATLG),
//             DCB=(BLKSIZE=23440),...
```

As a result, the binder uses a 23440-byte block size for the program.

This option is only valid when processing load modules.

DCBS=NO is the default value and can also be specified with the keyword **NODCBS**.

DYNAM: DYNAM option

If DYNAM(DLL) is enabled and the module contains exported symbols, the binder will build the control structures enabling the output module to be used as a DLL. The functions or variables exported by the DLL can be imported by DLL applications. If DYNAM(DLL) is enabled, and the module contains symbols eligible for dynamic resolution, and these symbols match symbols on IMPORT control

statements, then the binder will build the control constructures enabling the output module to execute as a DLL application. A DLL application can use functions or variables exported by DLLs.

You can specify the DYNAM option in the PARM field as follows:

```
DYNAM={DLL | NO}
```

Notes:

1. When DYNAM (DLL) is specified, a side file of IMPORT control statements might be generated by the binder.
2. If you are using the batch interface of the binder, the IMPORT control statements are saved in the data set specified in the SYSDEFSD ddname in your JCL. See "SYSDEFSD DD statement" on page 37. If you are using the binder API, the side file is saved in the data set represented by the SIDEFILE specification of the *files* parameter of the STARTDialog API. For more information, see *z/OS MVS Program Management: Advanced Facilities*.
3. A module linked with the DYNAM(DLL) option will be saved in a PO3 format program object unless you specify a higher COMPAT option or other features that force saving in an alternate format program object.
4. The DYNAM option disables the RES option.

EDIT: Edit option

To prevent a module from being reprocessed by the binder or linkage editor, you can mark it as not-editable. To assign the not-editable attribute, code **NE** or **EDIT=NO** in the PARM field.

```
{EDIT | NE | EDIT=NO}
```

EDIT is the default value.

If you use the not-editable attribute for a load module, you cannot request an EXPAND operation on the output module. You can only use AMASPZAP 18 consecutive times.

If you use the not-editable attribute for a PM1 format program object, you cannot use the EXPAND control statement.

If you use the not-editable attribute for a PM2 or higher format program object, there are the following additional restrictions:

1. You cannot use the EXPAND control statement.
2. You cannot run AMASPZAP against it.
3. You cannot list the module with AMBLIST.
4. You canot process the module with the DLLRNAME utility.
5. You cannot copy the module to a PDS.
6. You cannot access the module using the binder API.
7. You cannot process the module with IEWTPORT or IEWBFDA.

A PM2 or higher format program object created with the not-editable option may require much less space on DASD. The size of the loaded program and the time taken to load the program will not change.

If you use the not-editable attribute when creating a program object which would meet the limitations of PM3 or lower format, except that it contains symbols longer than 1024 bytes, the object will be given execution attributes equivalent to a PM3 object. This will allow it to be executed on down-level systems. See the “COMPAT: Binder level option” on page 74 for additional information.

EP: Entry point option

The EP option allows you to specify an external name to be used as the entry point for the program. The EP option is overridden by the ENTRY control statement. You can specify up to 1024 characters for the name but the JCL PARM field is limited to 100 characters and an OPTIONS data set is limited to 80 characters per option, including the “EP=”.

Specify the EP option on the PARM statement as follows:

EP=*name*

EXITS: Specify exits to be taken option

The EXITS option allows you to specify an exit(s) to be taken during binder processing. For more information, see *z/OS MVS Program Management: Advanced Facilities*.

EXITS=(*exit*(*module-name[,variable]*),...)

where

exit
: Specifies the user exit(s) to be selected. Choose one or more user exit names from INTFVAL, MESSAGE, and SAVE.

module-name
: Specifies the name of your loadable exit module

variable
: Specifies an optional variable to be passed to your exit routine as follows:

  For the INTFVAL exit you can specify an option string of up to 64 characters (if the string is enclosed in quotes, the quotes are removed).

  For the MESSAGE exit you can specify one numeric value (specify 4 to suppress the message or zero to allow the message to be printed).

EXTATTR: Specify extended attributes

The EXTATTR option allows you to set extended attributes for SYSLMOD when saved in a z/OS UNIX file.

Four extended attributes can be set:

1. APF authorization
2. PGMCNTL

3. NOSHAREAS
4. SHRLIB

EXTATTR={*suboption*	(*suboption*[,*suboption*]...)}

Where 'suboption' can be any of the following keywords:

APF | NOAPF | SHAREAS | NOSHAREAS | PGM | NOPGM | SHRLIB | NOSHRLIB

Up to four suboptions can be given in a single EXTATTR specification. The last valid specification for each of the four bits takes precedence. The defaults for the files are ordinarily NOAPF, SHAREAS, NOPGM and NOSHRLIB. The binder will not attempt to change the system settings for any attribute for which the user has not specified a value.

APF
Causes the APF authorized flag for the SYSLMOD file to be set.

NOAPF
Will cause the flag to be set off.

PGM
Will cause the program-controlled flag for the SYSLMOD file to be set.

NOPGM
Will cause the flag to be set off.

SHAREAS
Will cause the NOSHAREAS attribute flag for the SYSLMOD file to be turned off

NOSHAREAS
Means that the flag is set on

SHRLIB
Will cause the SHRLIB attribute for the SYSLMOD file to be turned on

NOSHRLIB
Will cause the SHRLIB attribute to be turned off

For further information on the extended attributes, refer to *z/OS UNIX System Services Command Reference*.

FETCHOPT: Fetching mode option

The FETCHOPT option allows you to specify how a program object should be *paged-mapped* (loaded) into virtual storage for execution. The syntax of the FETCHOPT option is:

FETCHOPT={(PACK,PRIME) | (NOPACK,PRIME) | (NOPACK,NOPRIME)}

PACK | NOPACK
Allows you to specify whether the program object is page-mapped into virtual storage on a page or double word boundary. Specifying **PACK** causes the program object to be page-mapped into page-aligned virtual storage and then moved to storage with double word alignment.

Specifying the **NOPACK** suboption of FETCHOPT will mark a program object as eligible to be page-mapped into page-aligned virtual storage without a secondary move. Other characteristics of the program, in conjunction with loading algorithms designed to optimize performance or storage usage, may prevent this loading method from actually being used.

PRIME | NOPRIME
Allows you to specify if the program object should be completely read into virtual storage before execution. When **PRIME** is coded, all of the program pages are read before program execution begins. When **NOPRIME** is coded, program pages are not read until they are needed during execution.

You cannot specify the combination **(PACK,NOPRIME)**. The default is **(NOPACK,NOPRIME)**.

This option is only valid when processing program objects.

When a program object is loaded from a z/OS UNIX file, it is not page-mapped. NOPRIME is ignored and the entire program is read in before program execution begins. Specifying the PACK option for a program object loaded from a z/OS UNIX file results in doubleword alignment, but does not result in a secondary move.

FILL: Fill character option

The FILL option lets you specify the character to be used to fill uninitialized areas of the program object.

FILL={*byte* | **NONE**}

The value *byte* (two hexadecimal digits) is used to specify a byte value that is used to fill uninitialized areas of the program object. All of the hexadecimal (X'00'-X'FF') values are valid. For example, FILL=81 fills the area with X'81'.

The FILL option has no effect on storage added by the EXPAND statement. It also has no effect on load modules and PM1-format program objects.

GID: Specify group ID

The GID option allows you to specify the Group ID attribute to be set for the SYSLMOD file:

GID=*value*

where

value
A string of up to 8 alphanumeric characters that represents a group name or numeric z/OS UNIX group id. The characters will be folded to uppercase unless 'value' is enclosed in quotes.

HOBSET: Set high order bit option

The HOBSET option allows you to specify if the high order bit in each four byte V-type address constant is set according to the AMODE of the target symbol.

```
HOBSET={NO | YES}
```

YES
Specifies the high order bit in each V-type address constant is set according to the AMODE of the target entry point. For AMODE(31) or AMODE(ANY) targets, the high order bit is set on (B'1'). If the target is marked AMODE(64), the address constant will not be altered. For AMODE(24), the high order bit is set off.

Note: This operation is completely reversible. On rebinding, V-cons from included program objects revert to their original state, unless HOBSET is specified again.

NO
Specifies the high order bit in each V-type address constant is not to be set according to the AMODE of the target entry point.

NO is the default. The bit is set to off if HOBSET is not specified from any source.

Note: A module or element loaded below 16 MB might need to operate with AMODE(31) if it receives control from another module or element loaded above 16 MB. This allows it to access the caller's data areas.

LET: Let execute option

Ordinarily, the binder marks an output program module as nonexecutable when an error with a severity level of 8 or higher is encountered. You can override this by specifying a different severity level using the LET option. The binder then marks the module as not-executable only if an error is encountered whose severity level is higher than what you specified.

Specify the LET option by coding the PARM field as follows:

```
{LET={0 | 4 | 8 | 12} | NOLET}
```

LET=4 is the default value. Coding the **NOLET** keyword will cause the binder to mark the output module as nonexecutable when an error occurs with a severity level of 4 or higher. If LET is specified without a value, LET(8) is assumed.

If **LET=4** is specified, **XCAL** does not need to be specified.

LINECT: Line count option

The LINECT option lets you specify the number of lines to be included on each page of binder output listings, including header lines and blank lines. The LINECT option is coded in the PARM field as follows:

```
LINECT={0 | 60 | n}
```

The value *n* can be any integer between 24 and 200, or 0. If you specify 0, there are no page breaks in the output listing. The default value is **LINECT=***60*.

LIST: Listing option

The LIST option allows you to control the type of information included in the SYSPRINT or SYSLOUT data set. Consult Chapter 8, “Interpreting binder listings,” on page 127 for an explanation and examples of the various kinds of information available. Code the LIST option in the PARM field as follows:

```
{LIST | LIST={ALL | SUMMARY | STMT | NOIMP[ORT] | OFF} | NOLIST}
```

The LIST value can be one of the following:

ALL
Produces a listing of individual function calls, the load or save summary, control statements, and messages. Messages IEW2308I and IEW2413I are issued only if LIST=ALL.

SUMMARY
Produces a listing of the load or save summary (including processing options and module attributes), control statements, and messages.

STMT
Produces a listing of control statements and binder messages.

NOIMPORT | NOIMP
Produces the same output as SUMMARY except IMPORT control statements are not echoed in message IEW2322I.

OFF
Produces a listing that contains only binder messages.

LIST=SUMMARY is the default value. The keyword **LIST** is equivalent to **LIST=SUMMARY**. **NOLIST** is equivalent to **LIST=OFF**.

LISTPRIV: List unnamed sections option

The LISTPRIV option allows you to obtain a list of unnamed (’private code’) sections. Unnamed sections are sections which were input to the Binder with no name (that is, the name consists of all blanks). The use of unnamed sections is not recommended (They may cause code growth on rebinding and may create maintenance problems.) LISTPRIV is useful as a tool in locating such sections in your binds.

```
LISTPRIV={NO | YES}
```

YES
If unnamed sections exist, a level 8 error message will be generated, and a report which lists all such sections and their origins will be produced. If no unnamed sections exist, LISTPRIV will have no effect..

NO
No diagnostics or special reports will be generated for unnamed sections.

NO is the default.

MAP: Program module map option

The binder allows you to request a program module map by coding **MAP** in the PARM field as follows:

```
{MAP | MAP=NO | NOMAP}
```

When the **MAP** option is specified, the binder produces a map of the program module in the diagnostic data set SYSPRINT, SYSLOUT, or SYSTERM. In the case of an empty module, no program module map will be generated. Figure 25 on page 131 contains an example of a program module map.

Starting in z/OS V1.6, when a bind specifying the **MAP** option fails resulting in a not-executable (NX) module, a program module map will be included in the binder listing.

MAP=NO is the default value and can also be specified with the keyword **NOMAP**.

MAXBLK: Maximum block size option

You can specify the maximum size of a text block within a load module by coding the MAXBLK option in the PARM field as follows:

```
MAXBLK=n
```

The MAXBLK value *n* specifies the length of the text block in bytes and must be an integer between 256 and 32760. This option allows you to ensure that a load module can be copied to a device with a smaller track size without reblocking.

If you specify *value2* on the SIZE option but do not specify a MAXBLK value, MAXBLK will default to one-half of *value2*. If you do not specify either value, MAXBLK defaults to the block size of the data set. If you code the DC option, MAXBLK and SIZE are both overridden and MAXBLK is set to 1024 bytes.

We recommend that you allow the system to determine the block size for program libraries. However, if you need to control the block size, we recommend that you use the MAXBLK option instead of the SIZE option.

This option is only valid when binding load modules.

MSGLEVEL: Message level option

The binder allows you to limit the messages displayed to only those of a specified severity level and higher. You specify this level by coding the MSGLEVEL option in the PARM field as follows:

```
MSGLEVEL={0 | 4 | 8 | 12}
```

The MSGLEVEL value is a message severity level. The default value is **MSGLEVEL=***0*.

NAME: NAME option

The NAME option allows you to specify a name to be used to identify a loaded program to the system. You can specify the NAME option only when you are using IEWBLDGO.

You specify the NAME option on the PARM statement as follows:

NAME=*name*

The maximum length for the name is 8 characters.

The default value for this option is ***GO*.

OL: Only-loadable option

The only-loadable option lets you specify that a module can only be brought into virtual storage using a LOAD macro instruction.

A module with the only-loadable attribute must be entered with a branch instruction or a CALL macro instruction. If an attempt is made to enter the module with a LINK, XCTL, or ATTACH macro instruction, the program making the attempt is terminated abnormally by the control program. (See *z/OS MVS Programming: Assembler Services Guide* for information on the LINK, XCTL, and ATTACH macro instructions.)

You specify the only-loadable option in the PARM field as follows:

{OL | OL=NO | NOOL}

OL=NO is the default value and can also be specified with the keyword **NOOL**.

OPTIONS: Options option

Instead of providing all processing options in the PARM field, you can create a data set containing the options. You specify the ddname of the data set by coding the OPTIONS option in the PARM field as follows:

OPTIONS=*ddname*

ddname identifies a sequential data set of blocked or unblocked 80-byte records. Options are specified just as they are in the PARM field, separated by commas. Option records cannot be continued. A blank outside of a quoted string ends processing of options in that record.

The options data set can contain multiple records with individual parameter sets. It cannot contain the **OPTIONS** option or any of the Environmental options (see Table 6 on page 69. Blank records are ignored. See “Options data set” on page 34 for information on coding the DD statement that defines the options data set.

Tip: The options file does not replace the options string, but instead treats it as if the file was inserted into the options string at the point where the OPTIONS option appears.

OVLY: Overlay option

The OVLY option allows you to create a program module in overlay format. A program with the overlay attribute is placed in an overlay structure as directed by binder OVERLAY control statements. The program module cannot be refreshed, reenterable, or serially reusable. AMODE(24) and RMODE(24) are the only valid addressing and residence options.

If the overlay attribute is specified and no OVERLAY control statements are found in the binder input, the attribute is ignored.

The overlay attribute must be specified for overlay processing. If this attribute is omitted, the OVERLAY and INSERT statements are not considered valid, and the module is not put into overlay structure.

You specify the overlay attribute by coding **OVLY** in the PARM field as follows:

{OVLY | OVLY=NO | NOOVLY}

See Appendix D, "Designing and specifying overlay programs," on page 193, for information on the design and specification of an overlay structure.

OVLY=NO is the default value and can also be specified with the keyword **NOOVLY**.

Note: The OVLY option overrides any specification of the COMPAT option. That is, if you specify the options COMPAT (COMPAT=any value) and OVLY at the same time, OVLY prevails and the module is saved in PM1 format if the SYSLMOD data set is a PDSE. Otherwise it is saved as a load module in a PDS. For more information on COMPAT, see "COMPAT: Binder level option" on page 74.

PATHMODE: Set z/OS UNIX file access attributes for SYSLMOD

PATHMODE is used to set z/OS UNIX files attributes for SYSLMOD.

PATHMODE=*oct1,oct2,oct3,oct4*

oct1,oct2,oct3,oct4
Where *oct1* through *oct4* are each are specified as an octal digit (0-7) separated by commas. Each of these digits specifies execution values that override the permission bits set by the PATHMODE parameter in the JCL for SYSLMOD.

The octal digit is interpreted as three bits (e.g. 5 is 101) and used as follows:

oct1

1.. Set user ID of process to user ID of file owner when the program is executed

.1. Set group ID of process to group ID of file owner when the program is executed

..1 Keep loaded executable in storage

oct2

1.. Owner permission to read file

.1. Owner permission to write file

..1 Owner permission to execute file

oct3

1.. Group permission to read file

.1. Group permission to write file

..1 Group permission to execute file

oct4

1.. Other permission to read file

.1. Other permission to write file

..1 Other permission to execute file

z/OS MVS JCL Reference and *z/OS UNIX System Services Command Reference* have more information on PATHMODE file access attributes.

PRINT: Diagnostic messages option

Informational and diagnostic messages are normally written to the SYSLOUT or SYSPRINT data sets. You can turn off this feature by coding **NOPRINT** in the PARM field.
If **NOPRINT** is coded, the SYSLOUT and SYSPRINT data sets are not opened.

{PRINT | NOPRINT}

RES: Search link pack area option

During IEWBLDGO processing, the binder automatically searches the link pack area queue before searching the SYSLIB data set. You can prevent this by coding the NORES option in the PARM field.

{RES | NORES}

NORES is the default for the bind and save entry point (IEWBLINK or its aliases). RES is the default for the batch load entry points.

REUS: Reusability options

The REUS option allows you to specify how a program can be reused. (Reusability means that the same copy of a program module can be used by more than one task either concurrently or one after another.)

Note that the value of the REUS option *always* overrides the reusability of any included load modules or program objects.

The syntax of the REUS option is as follows:

REUS={NONE | SERIAL | RENT | REFR}

The reusability values are:

NONE
The module cannot be reused. A new copy must be brought into virtual storage for each use. NONE is the default value.

SERIAL
The module is serially reusable. It can only be executed by one task at a time; when one task has finished executing it another task can begin. A serially reusable module can modify its own code, but when it is reexecuted it must initialize itself or restore any instructions or data that have been altered.

RENT
The module is reenterable. It can be executed by more than one task at a time. A task can begin executing it before a previous task has completed execution. A reenterable module cannot modify its own code. (MVS protects your module's virtual storage so that your module cannot be modified.)

Reenterable modules are also serially reusable.

REFR
The module is refreshable. It can be replaced by a new copy during execution without changing the sequence or results of processing. A refreshable module cannot be modified during execution.

A module can only be refreshable if all the control sections within it are refreshable. The refreshable attribute is negated if any input modules are not refreshable. Refreshable modules are also reenterable and serially reusable.

The refreshable attribute can be specified for any nonmodifiable module.

Alternatively, you can code a REUS option as a single keyword without a value (**REUS, NOREUS, RENT, NORENT, REFR, NOREFR**). For example:

```
//LKED  EXEC PGM=IEWBLINK,PARM='RENT,...'
```

REUS used as a single keyword is equivalent to REUS=SERIAL. NOREUS used as a single keyword is equivalent to REUS=NONE. This alternative form is supported only for backward compatibility. The most restrictive positive specification is used to set the reusability attribute. For example, specifying REFR has the same effect as specifying REUS (REFR) and the module is marked as refreshable, reenterable, and (serially) reusable.

If the PARM string contains both formats, the **REUS(***value***)** instance will override any reusability options specified without values.

The binder only stores the attribute in the directory entry. It does not check whether the module is actually reenterable or serially reusable. If the module is incorrectly marked as reenterable or reusable, execution results are unpredictable; for example, a protection exception might occur or the program might use another task's data.

RMODE: Residence mode option

To assign the residence mode for all the entry points into a program module, you can code the RMODE parameter as follows:

```
RMODE({24 | ANY | SPLIT})
```

The residence mode assigned in the PARM field is overridden by a residence mode assigned in the MODE control statement, but overrides the accumulated residence mode found in the ESD data for the control sections or private code in the input.

AMODE and RMODE values are specified independently, but checked for conflicts before output processing occurs. See “AMODE and RMODE combinations” on page 27 for information on AMODE and RMODE compatibility and the setting of default values.

RMODE(SPLIT) specifies the program text (class B_TEXT) can be split into two class segments according to the RMODE of each section. Rules for splitting the text are:

- If RMODE(SPLIT) is specified, the B_TEXT class of each included module is distributed between the two class segments according to the RMODE of each section contained in the module.
- If RMODE(SPLIT) is not specified, either through the binder execution parameter or a control statement, included text in classes B_TEXT, B_TEXT24 and B_TEXT31 are combined into B_TEXT class and loaded into memory using the existing RMODE resolution rules.
- If the OVLY option is specified, RMODE is reset to 24 and the split module is not produced.
- If RMODE(SPLIT) is specified, consider the HOBSET option. If you specify HOBSET, the high order bit of each V-type address is set according to the AMODE of the called entry point.

When an RMODE(SPLIT) module is loaded, the LOAD service returns a length of zero. For additional information on multiple segment modules, see “Creating a program object” on page 21. When you use LOAD, the CSVQUERY service should be used with the OUTXTLST parameter to obtain information about the address (load point) and length of each program segment. See CSVQUERY in *z/OS MVS Programming: Assembler Services Guide* for more information.

The keyword **RMODE** can also be specified as **RMOD**.

SCTR: Scatter load option

SCTR causes special control tables to be built in the output load module. This information is used by the system when loading the nucleus. Otherwise the tables are ignored. The option applies only when saving a load module.

The syntax of the SCTR option is as follows:

```
SCTR={NO | YES}
```

The default is NO.

SCTR or SCTR=YES must be specified when building a module that represents the system nucleus.

SIZE: Space specification option

The SIZE option allows you to specify the amount of space available for processing load modules. You can specify the amount of virtual storage the binder can use and the size of the load module buffers. If you specify SIZE when you bind program objects, the *value* subparameter is ignored. Also, if you specify WKSPACE, the first subparameter of WKSPACE overrides the first subparameter of SIZE.

The syntax of the SIZE option is:

SIZE={*value1*[**K**] | ([*value1*[**K**],*value2*[**K**])}

value1
Specifies the maximum number of bytes of available virtual storage. For the binder, the minimum value is 16 KB (16384) and the maximum value is 16000 KB (16 MB).

value2
Specifies the number of bytes of storage to be allocated for the load module buffer. For the binder, the minimum value is 512 and the maximum value is 65520 (approximately 64KB).

The binder only uses this value to determine the block size of the load module. If MAXBLK is not specified, the block size is set to half of *value2*.

When coded in the PARM field, *value1* and *value2* parameters are enclosed in parentheses. For example:

```
//LKED       EXEC   PGM=IEWBLINK,PARM='SIZE=(2048K,32K),...'
```

Both *value1* and *value2* can be expressed as integers specifying the number of bytes of virtual storage or as nK, where “n” represents the number of 1KB (1024) of virtual storage.

The binder provides default values for the SIZE option. The default values are used if you do not specify any values, or if you specify one or more of the values incorrectly. These defaults should be adequate for most binds, relieving you from needing to specify the SIZE option. We recommend that you do not use the SIZE option. Block size for load modules should be specified with the MAXBLK option (see “MAXBLK: Maximum block size option” on page 84), and workspace can be allocated with the WKSPACE option (see “WKSPACE: Working space specification option” on page 93).

SSI: System status index option

You can specify hexadecimal information to be placed in the system status index by coding the SSI option in the PARM field as follows:

SSI=*ssi-info*

ssi-info is a hexadecimal value of exactly 8 digits. This is placed in the system status index of the output module library directory entry.

If a SETSSI control statement has been coded, the value specified there overrides any value set by this option.

STORENX: Store not-executable module

Normally, the binder does not replace an executable module in a program library with a not-executable version. You can override this standard action with the STORENX option.

{STORENX | STORENX=NO | NOSTORENX}

When the STORENX option is coded, a new module replaces an existing module of the same name regardless of the executable status of either module. If the NAME statement is provided, the replace option (**R**) must have been coded.

STORENX=NO is the default value and can also be specified with the keyword **NOSTORENX**.

TERM: Alternate output option

You can request that the numbered error and warning messages be written to the data set defined by a SYSTERM DD statement by coding **TERM** in the PARM field.

{TERM | TERM=NO | NOTERM}

When the TERM option is specified, a SYSTERM DD statement must be provided. If it is not, the TERM option is ignored and messages are written only to the SYSPRINT or SYSLOUT data set.

Output specified by the TERM option supplements printed diagnostic information. When TERM is used, binder error/warning messages appear in both output data sets.

TERM=NO is the default value and can also be specified with the keyword **NOTERM**.

TEST: Test option

A program with the test attribute contains information about internal symbols in a form that can be accessed with the TSO TEST command. Symbol tables to be used by the TSO TEST command should be included in the input to the binder, which will place them in the output module. If the test attribute is not specified, any symbol tables in the input are ignored by the binder and are not placed in the output module. If the test attribute is specified, and no symbol table input is received, the output load module will not contain symbol tables to be used by the TSO TEST command.

Specifying the TEST option is not useful unless you are going to use the TSO TEST command on the program. The symbol tables in the program are ignored except when using the TSO TEST command.

You assign the test attribute by coding TEST in the PARM field.

{TEST | TEST=NO | NOTEST}

The TEST option is only valid for program modules that are stored in a program library for later execution.

TEST=NO is the default option and can also be specified with the keyword **NOTEST**.

TRAP: Error recovery

Specifying the TRAP option lets you control error trapping.

This option can be specified only in the following ways:

- The PARM string when the binder is invoked from JCL.
- The first parameter in the parameter list passed when calling the binder from another program (IEWBLINK, IEWBLOAD, IEWBLODI, IDWBLDGO).
- The IEWBIND API FUNC=STARTD OPTIONS= or PARMS= parameters.

{TRAP=ON | ABEND | OFF}

ON
Causes the binder to establish both an ESTAE and an ESPIE exit. This will trap all abends and program checks that occur while the binder is in control. A key aspect is that parameter validation done by the binder API will return the documented results even if some program in the binder calling sequence has a program check exit.

ABEND
The binder will establish an ESTAE exit but not an ESPIE exit. This will trap all abends, but program checks will be caught by the binder only if no program in the binder calling sequence has an ESPIE exit.

Notes:

1. Especially with the API interface, program checks may occur during binder validation of its input. The binder will normally recover from those and convert them into return codes. It will be unable to do that if TRAP=ABEND was specified and some calling program has an ESPIE exit.
2. An LE environment will normally include an ESPIE exit, so LE-enabled programs calling the binder should not use TRAP=ABEND unless they are being debugged or have made special provision for this situation.
3. Prior to z/OS 1.5 there was no TRAP option, but the binder behavior matched what is now defined for TRAP=ABEND.

OFF
Prevents the binder from establishing any ESTAE or ESPIE exit. This will allow callers of the binder to trap all abends and program checks.

Note: Many data set related ABENDs are passed directly by DFSMS to binder routines doing I/O. These do not go through binder ESTAE processing and will continue to be caught even with TRAP=OFF.

UID: Specify user ID

The UID option allows you to specify the User ID attribute to be set for the SYSLMOD file:

UID=*value*

where

value
A string of up to 8 alphanumeric characters that represents a user name (such as TSO logon ID) or a numeric z/OS UNIX user id.

UPCASE: UPCASE option

This option indicates whether additional renaming should be done when symbols remain unresolved. Unresolved function references that are marked as renameable and are not imported are set to uppercase if they are eight characters or less in length. Also, underscore (`'_'`) is mapped to `'@'` and names beginning with IBM, CEE, or PLI have their respective prefixes changed to IB$, CE$, and PL$. After the renaming process is complete, an attempt to resolve the symbols using the new names is made. Traditional object modules do not support the renameable bit and thus symbols originating from them are not affected by the UPCASE option.

The UPcase option provides binder function roughly equivalent to the prelinker UPCASE option.

The UPCASE option can be specified in the PARM field as follows:

{UPCASE | UPCASE=YES | UPCASE=NO | NOUPCASE}

Note: UPCASE is supported only for format 3 or higher program objects. This is expressed as COMPAT=PM3 or equivalent, or higher. But when COMPAT=MIN is indicated, the binder does not force PM3 or higher simply to satisfy UPCASE=YES.

WKSPACE: Working space specification option

The WKSPACE option allows you to specify the amount of virtual storage available to the binder during processing.

The syntax of the WKSPACE option is:

WKSPACE=([*value1***][,***value2***])**

value1
The maximum amount of virtual storage below the 16 MB line, in units of 1KB, that is available for binder processing.

value2
The maximum amount of virtual storage above the 16 MB line, in units of 1KB, that is available for binder processing.

For example:

```
//LKED       EXEC   PGM=IEWBLINK,PARM='WKSPACE=(96,1024),...'
```

If *value1* is not specified and the SIZE option has been specified, *value1* is set to *value1* as specified on the SIZE option. If the SIZE option is not specified, the binder assumes that it can use all available virtual storage below 16 MB. We recommend that you use the WKSPACE option with the MAXBLK option and in place of the SIZE option.

If *value2* of the WKSPACE option is not specified, the binder allocates workspace from above 16 MB as needed until no more space is available.

Under normal circumstances, the binder can determine its own workspace requirements. You should not need to specify the WKSPACE parameter unless you have unusual virtual storage considerations.

We recommend a minimum of 96 KB below 16 MB and 2048 KB above 16 MB for all binder processing.

XCAL: Exclusive call option

You use the XCAL option when valid exclusive references have been made between segments of an overlay program. A warning message is issued for each valid exclusive reference, but the binder marks the output module as executable.

See “References between segments” on page 198 for information about valid exclusive references.

To specify the exclusive call option, code XCAL in the PARM field.

{XCAL | XCAL=NO | NOXCAL}

The OVLY attribute must also be specified when you use the XCAL option. For example:

```
//LKED       EXEC   PGM=IEWBLINK,PARM='XCAL,OVLY,...'
```

XCAL=NO is the default value and can also be specified with the keyword **NOXCAL**.

XREF: Cross reference table option

You can request a cross-reference table of a program module by coding XREF in the PARM field.

{XREF | XREF=NO | NOXREF}

When the XREF option is specified, the binder produces a cross-reference table of the program module in the SYSPRINT data set. In the case of an empty module, no program module map will be generated. If you also need a module map, you must request one using the **MAP** option. Figure 28 on page 136 contains an example of a cross reference table.

Starting in z/OS V1.6, when a bind specifying the **XREF** option fails resulting in a not-executable (NX) module, a cross-reference table will be included in the binder listing.

XREF=NO is the default value and can also be specified with the keyword **NOXREF**.

Chapter 7. Binder control statement reference

You provide control statements to the binder to specify editing operations and identify additional input. You can provide entry and module names and specify the authorization code of a program module.

This chapter summarizes the binder control statements. Statement descriptions are in alphabetical order, and include the purpose, syntax, placement in the input stream, and examples.

Before using these control statements, you should also be familiar with the syntax and national conventions described in “Notational conventions” on page xvi.

Note: This chapter refers to binder processing. These concepts apply equally to linkage editor and batch loader processing unless noted otherwise in Appendix A, “Using the linkage editor and batch loader,” on page 157. The linkage editor and batch loader cannot process program objects.

Binder syntax conventions

Each binder control statement specifies an *operation* and one or more *operands*. Nothing must be written preceding the operation, which must begin in or after column 2. The operation must be separated from the operand by one or more blanks; blanks cannot be embedded within the operand field (see “Rules for comments” on page 99).

Control statements are specified in 80-byte lines. A control statement can be continued on as many lines as necessary. However, the control statement keyword must be entirely on the first line and the operands must begin on the first line. A control statement can be continued in one of the following ways:

1. Terminate an operand at a comma followed by a blank. The comma must be in column 71 or earlier. Continuation lines can begin anywhere after column 1. Any leading blanks are discarded.
2. If the operand field goes to column 71 (with no embedded blanks) and column 72 is nonblank, the next line is treated as a continuation line. As in 1, the continuation line can begin anywhere after column 1 and any leading blanks are discarded. Columns 73 through 80 of each line are reserved for sequence numbers, which are not processed by the binder.
3. An operand enclosed in single quotation marks can be continued. The binder searches as many records as necessary until it finds the ending quote. The full operand is reconstructed by concatenating the fragments starting with column 2 of each line. In this case, the continuation of the operand must start in column 2, or the operand is considered to have embedded blanks and is truncated at the first blank. You can continue coding additional operands as usual following the ending quote.

Most binder control statements require various symbols or names to be specified as operands. Unless otherwise noted, all such names and symbols must be 32767 bytes or less and consist of EBCDIC characters within the range of X'41' through X'FE' plus the double-byte character set (DBCS) SO/SI control characters X'0E' and X'0F'. It is strongly recommended that all such names consist of displayable characters only and that they are enclosed by single quotation marks if they contain

other than alphanumeric or national characters. DDnames, member names, and alias names must conform to the JCL coding rules for those parameters and can not exceed 1024 bytes in length.

You can enclose any symbol except binder-defined keywords with single quotation marks. If you want to use commas or parentheses in a symbol in a control statement, you must enclose that symbol in single quotation marks. A single quotation mark embedded in a quoted string must be coded as two consecutive quotation marks. Only complete symbols can be enclosed in single quotation marks. Characters within quoted strings will not be folded to upper case, regardless of the value of the CASE option. A quoted string with no closing quote continues in column 2 of the next line.

A number of metasymbols dealing with names and program symbols have been used in the control statement syntax diagrams in this chapter. These metasymbols include the following:

- *symbol, newsymbol*. A user-assigned name with a maximum length of 32767 bytes and consist only of characters from the binder's character set, described above.
- *externalsymbol, external reference*. Those symbols that are or will be defined in the External Symbol Dictionary (ESD). These include entry names defined by a Label Definition (LD), section names that are implied entry names, external references (ER) and part references (PR), also called pseudoregisters.
- *sectionname*. Those symbols which name sections in the module. Section is a generic term encompassing control sections, private code sections and common areas. Blank common and private code sections cannot be named on binder control statements.
- *directoryname*. Those symbols that appear or will appear in the directory of a named library structure. Directory names include member names, aliases and unqualified z/OS UNIX file names, and have length restrictions imposed by the underlying file system.

File system	Member name	Alias name
PDS Library	8	8
PDSE Library	8	1024
z/OS UNIX Directory	255	255

- *ddname*. The name coded in the label field of a dd-statement. Ddnames are limited to eight bytes.
- *pathname*. A z/OS UNIX pathname designating either a directory or a regular file (depending on the control statement). It must begin with either ″./″ (meaning a relative pathname) or ″/″ (meaning an absolute path name) and is limited to 1023 bytes in length. To prevent the pathname from being folded to uppercase, you should either enclose the pathname in single quotation marks or specify the binder CASE=MIXED option. z/OS pathnames are replaced in the binder listing output by generated ″ddnames″ of the form ″/nnnnnnn″, where nnnnnnn is numeric. The true pathname may be found in the DDname vs Pathname report.

You can include blank lines between control statements but not within a statement. A blank line indicates an end to any statement.

For more information on syntax and notational conventions, see “Notational conventions” on page xvi.

Syntax errors

If a syntax error is detected while processing a control statement, the remainder of the statement is skipped and not processed. However, any operands in the portion of the statement preceding the error are processed.

Rules for comments

Placing an asterisk (*) in column 1 of a control statement causes the binder to treat that line as a comment. The content of column 72 is ignored on a comment line. You can include comment lines anywhere in the control statement input except within a quoted string. You can also include comments on a control statement line; anything at the end of a control statement line separated from the operands by one or more blanks will be treated as a comment. Comments are not processed by the binder but can be printed.

A line is also treated as a comment if the previous statement ends with a blank but has a nonblank character in column 72.

Placement information

Binder control statements are placed before, between, or after object modules. They can be grouped, but they cannot be placed within a module. However, specific placement restrictions might be imposed by the nature of the services being requested by the control statement. Any placement restrictions are noted.

If a function can be specified either on a control statement or as an option in the PARM field of the EXEC statement, the control statement specification takes precedence.

ALIAS statement

The ALIAS statement specifies one or more additional names for the primary entry point, and can also specify names of alternate entry points.

Note: Alternate entry points are not supported for program objects that reside in z/OS UNIX files. If a z/OS UNIX path name is specified, that name becomes a true alias of the primary entry point.

The binder does not place a limit on the number of alias names that can be specified on an ALIAS statement or on separate ALIAS statements for one library member. These names are entered in the directory of the partitioned data set or PDSE in addition to the member name. If the symbol specified as the alias has appeared on an earlier ALIAS control statement, the new specification replaces the earlier one.

Note: If the module contains multiple text classes, all entry points must be defined in the same class.

The syntax of the ALIAS statement is:

ALIAS	{*directoryname*[(*externalsymbol*)]} {(**SYMLINK**, *pathname*)} {(**SYMPATH**, *pathname*)} [,...]

directoryname
Specifies an alternate name for the program object or load module. The symbol might or might not be the name of an external entry point within the program.

When the program is executed using the alias name, execution begins at the entry point associated with the alias. The entry point is determined according to the following rules:

1. If an *externalsymbol* is specified as an entry point (see below) for the alias, execution begins at that entry point.
2. If the alias symbol matches an entry name within the program, execution begins at that entry point.
3. If the alias symbol does not match an entry name within the program, execution begins at the main entry point.

externalsymbol
Specifies the name of the entry point to be used when the program is executed using the associated alias. If the external symbol is the name of an entry point within the program, that name is used as the entry point for the alias. If the external symbol is not an entry point name, but another external name such as a pseudoregister or an unresolved external reference, the main entry point is used as the entry point for the alias. If the symbol you specify is not defined in the program, the alias is not created.

SYMLINK
A symbolic link is a z/OS UNIX file that contains the pathname for another file or directory. Symbolic links can be links across mounted file systems.

SYMPATH
The contents of the path designated by a SYMLINK request are specified by the next following SYMPATH request.

pathname
The pathname to or contained by a symbolic link. The pathname contained in a symbolic link can be relative or absolute. If a symbolic link contains a relative pathname, the pathname is relative to the directory containing the symbolic link.

These entries can be repeated in any order. Alias entries can be divided up among separate ALIAS statements as desired except that there must be at least one SYMPATH specification following a given SYMLINK or group of SYMLINKs.

Placement: An ALIAS statement can be placed before, between, or after object modules or other control statements. It must precede a NAME statement used to specify the member name, if one is present.

Notes:

1. In an overlay program, an external name specified by the ALIAS statement must be in the root segment. In a multitext class program object, an alternate entry point specified by an ALIAS statement must be defined in the same class as the primary entry point.
2. When a program module is reprocessed, all ALIAS statements should be respecified so that the directory is updated. Otherwise, for replaced load modules, the aliases remain in the directory and point to the old library member. When a program object is replaced, the aliases are deleted.
3. Each alias name that is specified must be unique within the library. If the specified alias name matches an existing member name within the library, the alias will be rejected. If the specified alias name matches an existing alias name

in the library and the replace option **(R)** was not specified, the alias will be rejected. If replace was specified, the new alias name will replace the existing one.

4. To avoid name conflicts, delete obsolete alias names from the program library directory.
5. You can execute a program object that resides in a z/OS UNIX file by specifying an alias name. However, execution will always begin at the main entry point. By using the binder call interface, it is possible to copy the program module and its aliases to a partitioned data set or a PDSE. The alias information that was saved in the program object will be used to create aliases for the copied module as either true aliases or alternate entry points, in accordance with the rules documented here.

Symbolic link support

The SYMLINK and SYMPATH functions of the ALIAS control statement can be used to establish an arbitrary number of symbolic links. The contents of the path designated by a SYMLINK request are specified by the next following SYMPATH request. The result of a SYMLINK/SYMPATH pair is the creation of a file whose:
1. pathname is the SYMLINK path concatenated to the SYSLMOD path
2. file type is 'symbolic link'
3. contents are given by SYMPATH.

A SYMPATH specification applies to all SYMLINK specifications that precede it, back to the first previous SYMPATH.

Thus, in the following skeleton example:

```
ALIAS (SYMLINK,A1)
ALIAS (SYMLINK,A2)
ALIAS (SYMPATH,B1)
ALIAS (SYMLINK,A3)
ALIAS (SYMLINK,A4)
ALIAS (SYMLINK,A5)
ALIAS (SYMPATH,B2)
ALIAS (SYMLINK,A6)
```

SYMPATH B1 is used for A1 and A2, SYMPATH B2 is used for A3 through A5, and A6 is in error. Continuation rules and general syntactical rules are the same as those for other Binder control statements and control statement operands. Length limits for both the control statement and ADDA API call are 1024 for both SYMLINK and SYMPATH.

If the GID or UID options are specified, the UID and GID values for SYSLMOD are also used for the symbolic links.

Example

An output module, ROUT1, is assigned an alternate entry point, CODE1. CODE1 can also be invoked by an alias, CODE2. In addition, calling modules have been written using both ROUT1 and ROUTONE to refer to the output module. Rather than correct the calling modules, an alternate library member name is also assigned.

```
ALIAS      CODE1,CODE2(CODE1),ROUTONE
NAME       ROUT1
```

Because CODE1 is an entry name in the output module, execution begins at the point referred to when this name is used to call the module. The same entry point

will be selected when CODE2 is called, since CODE2 is an alias for the CODE1 entry point. The modules that call the output module with the name ROUTONE now correctly refer to ROUT1 as its main entry point. The names CODE1, CODE2, and ROUTONE appear in the library directory along with ROUT1.

AUTOCALL statement

The AUTOCALL control statement prompts the binder to perform incremental (or immediate) autocall using only the given library as the search library to resolve symbol references. See "Resolving external references" on page 49 for more information on autocall.

The syntax of the AUTOCALL statement is:

AUTOCALL	*ddname* | *pathname*

ddname
Specifies the name of a DD statement that describes a PDSE program object library, a PDS library containing object modules or load modules, or a z/OS UNIX directory or archive library file.

pathname
Specifies the absolute or relative pathname for a z/OS UNIX directory or archive library file. See "Binder syntax conventions" on page 97 for a discussion of continuations and lower case letters.

Placement: The AUTOCALL control statement can be placed anywhere in the job stream or input data set.

Notes:

1. This statement can be specified at any time during primary and secondary input to the binder. However, if there are any references left unresolved after any number of AUTOCALL control statements, the binder does not diagnose them.
2. If no autocall (NCAL or CALL=NO) is in effect, incremental autocall is not performed. See Chapter 6, "Binder options reference," on page 67 for information on the CALL and NCAL option.
3. The AUTOCALL statement replaces one form of the LIBRARY statement which was supported by the LE prelinker but is not supported by the binder. (See "Binder extensions supporting the Language Environment" on page 29.)
4. No symbol renaming is done when the binder attempts to resolve references during incremental autocall.

Example

The following example shows how the AUTOCALL statement is invoked to immediately resolve references made available during a recent INCLUDE.

```
//OBJMOD     DD     DSNAME=PROJECT.TAXES,DISP=(OLD,DELETE),...
//LOADMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=OLD,...
//SYSLIB     DD     DSNAME=PROJECT.MAIN.LOADLIB,DISP=OLD,...
//SYSLIN     DD     *
  INCLUDE  OBJMOD
  AUTOCALL LOADMOD
   ⋮
/*
```

In the example, OBJMOD is included first, followed by an autocall request that uses the LOADMOD module library to resolve references. At this point, no attempt is

made to resolve references using SYSLIB, and unresolved references are not diagnosed. The binder waits until all input has been specified to do a final autocall. At that time, it attempts to resolve any outstanding references by searching SYSLIB. After final autocall, if any references remain unresolved, the binder states them in its messages.

CHANGE statement

The CHANGE statement causes an external symbol to be replaced by the symbol in parentheses following the external symbol. The external symbol to be changed can be a control section name, a common area name, an entry name, an external reference, or a pseudoregister. More than one such substitution can be specified in one CHANGE statement. The syntax of the CHANGE statement is:

```
CHANGE      externalsymbol(newsymbol)
            [,externalsymbol(newsymbol)]...
```

externalsymbol
The external symbol that is changed.

newsymbol
The name to which the external symbol is changed.

Placement: In the job stream or input data set, the CHANGE control statement must be placed before either the module containing the external symbol to be changed, or the INCLUDE control statement specifying the module. The scope of the CHANGE statement is across the next object module, load module, or program object.

Notes:

1. External references from other modules to a changed control section name or entry name remain unresolved unless further action is taken.
2. If both the original name and the new name specified for the external symbol are already defined in the output module, the new name is deleted from the module before the original name is changed. If the new name defines a control section, the original section with the same name will be deleted. The results received from the binder under this condition vary from the results received from the linkage editor.
3. When a REPLACE statement that deletes a control section is followed by a CHANGE statement with the same control section name, the results are unpredictable.
4. If a CHANGE statement is not followed by any included module, the binder issues a diagnostic message and ignores the change.
5. If a CHANGE statement appears in a module included from an automatic call library, it will be ignored if it is not followed by a module from the same member.

Examples

Change Control Section Name: Example 1

Two control sections in different modules have the name TAXROUT. Because both modules are to be bound together, one of the control section names must be changed. The module to be changed is defined with a DD statement named OBJMOD. The control section name could be changed as follows:

```
//OBJMOD     DD     DSNAME=PROJECT.TAXES,DISP=OLD,...
//SYSLIN     DD     *
  CHANGE  TAXROUT(STATETAX)
  INCLUDE OBJMOD
   ⋮
/*
```

As a result, the name of control section TAXROUT in module TAXES is changed to STATETAX.

Change Module References: Example 2

A program object or load module contains references to TAXROUT that must be changed to STATETAX. This module is defined with a DD statement named LOADMOD. The external references could be changed at the same time the control section name is changed:

```
//OBJMOD     DD     DSNAME=PROJECT.TAXES,DISP=(OLD,DELETE),...
//LOADMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=OLD,...
//SYSLIN     DD     *
  CHANGE TAXROUT(STATETAX)
  INCLUDE OBJMOD
  CHANGE TAXROUT(STATETAX)
  INCLUDE LOADMOD(INVENTRY)
   ⋮
/*
```

As a result, control section name TAXROUT in module TAXES and external reference TAXROUT in module INVENTRY are both changed to STATETAX.

ENTRY statement

The ENTRY statement specifies the symbolic name of the first instruction to be executed when the program is called by its module (member) name for execution or by an alias that does not match an executable external symbol. An ENTRY statement should be used whenever a module is reprocessed by the binder. The syntax of the ENTRY statement is:

ENTRY	*externalsymbol*

externalsymbol
Defined as either a control section name or an entry name in an input module.

Placement: An ENTRY statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module, if one is present.

Notes:

1. If you provide more than one ENTRY statement, the main entry point specified on the *last* statement is used.
2. In an overlay program, the first instruction to be executed must be in the root segment.
3. The external name specified must be a name associated with an instruction, not data, if the module is executed.
4. An entry point assigned by the ENTRY control statement overrides an entry point specified on an END statement of an object module. It also overrides an entry point specified as an EP option in the PARM field of an EXEC statement if you are using IEWBLOAD.

5. If you do not use an ENTRY statement, the entry point is the one specified on the first object module END record processed that specifies an entry point. If no entry point is specified on an ENTRY statement, END record, or EP option for IEWBLOAD, the entry point is the first byte of the first control section in the program. If the module contains multiple text classes and an entry point is not specified, the results are unpredictable.
6. If the module contains multiple text classes, the primary and all alternate entry points must be defined in the same class.

Example

In the following example, the main entry point is INIT1:

```
//LOADLIB    DD     DSNAME=PROJECT.LOADLIB,DISP=OLD
//SYSLIN     DD     *
  ENTRY INIT1
  INCLUDE LOADLIB(READ,WRITE)
/*
```

INIT1 must be either a control section name or an entry name in one of the program objects or load modules named READ or WRITE.

EXPAND statement

The EXPAND statement lengthens control sections or named common areas by a specified number of bytes. The syntax of the EXPAND statement is:

EXPAND	*sectionname*(*length*[,*classname*]) [,*sectionname*(*length*[,*classname*])]...

sectionname
Symbolic name of a common area or control section whose length is increased.

length
The decimal number of bytes to be added to the length of the section. The length of the section can be expanded to reach the maximum text size of a program object or load module. The maximum text size of a program object is 1 GB; the maximum text size of a load module is 16 MB. Binary zeros are used to initialize an expanded control section.

classname
The name of the text class to be expanded. *Classname* is not valid when COMPAT=LKED or COMPAT=PM1. Classname defaults to B_TEXT if it is not specified.

A message indicates the number of bytes added to the control section and the offset, relative to the start of the control section, where the expansion begins. The *effective* length of the expansion is given in hexadecimal and can be greater than the *specified* length if, after the specified expansion, padding bytes must be added for alignment of the next control section or named common area.

Placement: An EXPAND statement can be placed before, between, or after other control statements or object modules. However, the statement must follow the module containing the control section or named common area to which it refers. If the control section or named common area is entered as the result of an INCLUDE statement, the EXPAND statement can appear anywhere between the INCLUDE and NAME statements.

Note: EXPAND should be used with caution so as not to increase the length of a program beyond its own design limitations. For example, if space is added to a control section beyond the range of its base register addressability, that space is unusable unless you make other changes to the program to allow it to address the extra space.

Example

In this example, EXPAND statements add a 250-byte patch area (initialized to zeros) at the end of control section CSECT1 and increase the length of named common area COM1 by 400 bytes.

```
//LKED       EXEC   PGM=IEWBLINK
//SYSPRINT   DD     SYSOUT=A
//SYSLMOD    DD     DSNAME=PROJECT.PROGLIB,DISP=OLD
//SYSLIN     DD     DSNAME=&&LOADSET,DISP=(OLD,PASS)
//           DD     *
  EXPAND             CSECT1(250)
  EXPAND             COM1(400)
  NAME               MOD1(R)
/*
```

IDENTIFY statement

The IDENTIFY statement specifies any data you supply be entered into the CSECT identification records (IDR) for a particular control section. The statement can be used either to supply descriptive data for a control section or to provide a means of associating system-supplied data with executable code. The syntax of the IDENTIFY statement is:

IDENTIFY	*sectionname*('*data*')[,*sectionname*('*data*')]...

sectionname
The symbolic name of the control section to be identified.

data
Specifies up to 80 EBCDIC characters of identifying information for program objects, and up to 40 characters for load modules. You can supply any information desired for identification purposes.

Placement: An IDENTIFY statement must follow the module containing the control section to be identified or the INCLUDE statement specifying the module.

The syntax rules for the operand field are:

1. Blanks are not allowed between the CSECT name and the left parenthesis.
2. No blanks or characters are allowed between the left parenthesis and the leading single quotation mark nor between the trailing single quotation mark and the right parenthesis.
3. The data field consists of from 1 to 80 characters for program objects and 1 to 40 characters for load modules; therefore, a null entry must be represented, minimally, by a single blank.
4. Blanks can appear between the leading single quotation mark and the trailing single quotation mark. Each blank counts as 1 character toward the character limit.
5. A single quotation mark between the leading quotation mark and the trailing quotation mark is represented by 2 consecutive quotation marks. The pair of quotation marks counts as 1 character toward the character limit.

6. The IDENTIFY statement can be continued. If you are using the binder, the *data* characters end in column 71 and continue at column 2 on the next line.
7. If a leading quotation mark is found, all characters are read in until a trailing quotation mark is found or the character limit is reached.
8. A blank following a comma that terminates an operand also terminates the operand field for that record.
9. Double-byte character set (DBCS) characters can be included within the descriptive data. DBCS characters must be delimited by the shift-out (X'0E') and shift-in (X'0F') characters. The shift-out and shift-in characters and the delimited DBCS characters count as one or two bytes, respectively, toward the total length of the string.

You can provide more than one IDENTIFY statement for each control section name when you are creating a program object. However, if you are creating a load module, you can provide only one IDENTIFY statement. If you provide more than one IDENTIFY statement per control section for load modules, the information on only the last IDENTIFY statement is saved. The contents of each IDENTIFY statement will be saved in a separate record in the program object.

Example

In this example, IDENTIFY statements are used to identify the source level of a control section, a PTF application to a control section, and the functions of several control sections.

```
//LKED       EXEC   PGM=IEWBLINK
//SYSPRINT   DD     SYSOUT=A
//SYSLMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=OLD
//OLDMOD     DD     DSNAME=PROJECT.OLD.LOADLIB,DISP=OLD
//PTFMOD     DD     DSNAME=PROJECT.PTF.OBJECT,DISP=OLD
//SYSLIN     DD     *

(input object deck for a control section named FORT)

  IDENTIFY   FORT('LEVEL 03')
  INCLUDE    PTFMOD(CSECT4)
  IDENTIFY   CSECT4('PTF99999')
  INCLUDE    OLDMOD(PROG1)
  IDENTIFY   CSECT1('I/O ROUTINE'),
             CSECT2('SORT ROUTINE'),
             CSECT3('SCAN ROUTINE')
/*
```

Execution of this example produces IDR records containing the following identification data:

- The component ID of the binder that produced the program object or load module, the binder version and modification level, and the date of the current binder processing of the module. This information is provided automatically irrespective of whether you specify an IDENTIFY statement.
- User-supplied data describing the functions of several control sections in the module, as indicated on the IDENTIFY statements.
- If the language translator used supports IDR, the identification records produced by the binder also contain the name of the translator that produced the object module, its version and modification level, and the date of compilation.

The IDR records created by the binder can be referenced by using the LISTIDR option of the service aid program AMBLIST. For instructions on how to use AMBLIST, see *z/OS MVS Diagnosis: Tools and Service Aids*.

IMPORT statement

The IMPORT statement specifies an external symbol name to be imported and the library member or z/OS UNIX file name where it can be found. An imported symbol is one that is expected to be dynamically resolved. The syntax of the IMPORT statement is:

IMPORT	{**CODE** | **DATA** | **CODE64** | **DATA64**},*dllname*,*import_name*[,*offset*]

{CODE | DATA | CODE64 | DATA64}
Mutually exclusive keywords that specify the type of symbol being imported.

If **CODE** or **CODE64** is specified, the *import_name* must represent the name of a code section or entry point. Specify **CODE64** when using 64-bit addressing mode or specify **CODE** for any other addressing mode.

If **DATA** or **DATA64** is specified, the *import_name* must represent the name of a variable or data type definition to be imported. Specify **DATA64** when using 64-bit addressing mode or specify **DATA** for any other addressing mode.

dllname
The name of the DLL module that contains the *import_name* to be imported. If it is a member of a PDS or PDSE, it must be a primary name or an alias. The length is limted to eight bytes unless it is an alias name in a PDSE directory. In that case, the limit is 1024 bytes. If it is a z/OS UNIX file, the file name is limited to 255 bytes.

import_name
The symbol name to be imported. In programming terms, it represents a function or method definition, or a variable or data type definition. This distinction is made by specifying either **CODE**, **CODE64**, **DATA**, or **DATA64**. The *import_name* can be up to 32767 bytes in length.

offset
Offset consists of up to 8 hexadecimal characters. The offset will be stored with the DLL information for an imported function. This is primarily for the use of LE (Language Environment).

In order to continue a *dllname* or an *import_name*, code a nonblank character in column 72. Either blanks or commas will be accepted as delimiters between parameters.

Placement: The IMPORT statement can be placed before, between, or after object modules or other control statements.

Notes:

1. The DYNAM(DLL) binder option must be specified for IMPORT statements to take effect (see Table 6 on page 69).
2. IMPORT statements are processed as they are received by the binder. However, symbol resolution is not done against the imported symbols until the binder's final autocall is finished.
3. A bind job for a DLL application should include an IMPORT control statement for any DLLs that the application expects to use. Otherwise, if the DLL name is unresolved at static bind time, it will not be accessible at run time.
4. Ensure that the *dllname* matches the actual name of the DLL. Otherwise, import names will not be resolved.

5. Typically, a dynamic link library will have an associated side file of IMPORT control statements, and you will include this side file when statically binding a module that imports functions or variables from that library. However, you can also edit the records in the side file or substitute your own IMPORT control statements so that some symbols are imported from DLLs in a different library.
6. Modules with imported symbols can be saved only in PM3 or later format.
7. When you rebind a DLL, you must include the IMPORT statements. Information from the IMPORT control statements is not retained from one bind to another if the object is stored as a PO1, PO2, or PO3 format program objects. If you rebind a PO4 or higher program objects, you may use the -IMPORTS options on the INCLUDE control statement to request that the IMPORT information saved from the previous bind be brought in. In this situation you do not need to include the IMPORT statements again.
8. Import control statements generated by the binder will contain quotes around both the symbol name and the DLL name.

Example

IMPORT statements specify which symbols should be imported from a DLL provider or providers:

```
//           EXEC PGM=IEWL,PARM='MAP,XREF,CASE=MIXED'
//LOADMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=SHR
//OBJECT1    DD     PATH='/sl/app1/pm3d3/dlla01',PATHDISP=(KEEP,KEEP)
//SYSLIN     DD     *
  IMPORT  CODE TAXES97,Compute_97_Taxes_Schedule1
  IMPORT  CODE TAXES97,Compute_97_Taxes_Schedule2
  IMPORT  CODE64 TAXES03,Compute_03_Taxes_Schedule1
  IMPORT  CODE64 TAXES03,Compute_03_Taxes_Schedule2
  IMPORT  DATA REVENUE,TotalRevenue
  IMPORT  DATA64 REVENUE03,TotalRevenue03
  INCLUDE OBJECT1
   ⋮
/*
```

In the example above, two 31-bit addressable functions from member TAXES97, two 64-bit addressable functions from member TAXES03, one 31-bit addressable data variable from member REVENUE, and one 64-bit addressable data variable from REVENUE03 are being imported. These members should be in a dynamic link library, which can be found by the system search mechanisms at execution time. For example, the dynamic link library containing these members could be part of the STEPLIB concatenation.

INCLUDE statement

The INCLUDE statement specifies sequential data sets, library members, or z/OS UNIX files that are to be sources of additional input for the binder. INCLUDE statements are processed in the order in which they appear in the input. However, the sequence of control sections within the output program object or load module does not necessarily follow the order of the INCLUDE statements. If the order of the CSECTs within the module is significant, you must specify the desired sequence by using ORDER statements. The syntax of the INCLUDE statement is:

INCLUDE	[{**-ATTR**, | **-IMPORTS**, | **-ALIASES**, | **-NOATTR**, | **-NOIMPORTS**, | **-NOALIASES**}...] {*ddname*[(*membername*[,...])][,...] | *pathname*[,...]}

Note: If options which contradict one another are specified, the last valid option specified will be used. For example, if both **-ATTR** and **-NOATTR** are specified in that order, the binder will honor the **-NOATTR** option.

-ATTR
Specifies that module attributes should be copied from the input module and be applied to the module being built by the binder.

Notes:

1. Attributes cannot be included if the input is an object module, or if there is no member name on the INCLUDE control statement and the INCLUDE designates a load module.
2. Attributes brought in for a given module specified with INCLUDE override attributes copied in for previous modules.
3. Attributes override attributes requested by the Binder invocation parameters, but not those set by control statements such as SETOPT or MODE.

-IMPORTS
Specifies that dynamic resolution information (if any) will be copied from the input module. Starting in z/OS V1.6 this option is no longer required, as the INCLUDE statement will always bring in any available dynamic resolution information unless it is suppressed by -NOIMPORTS. This option is still supported for compatibility reasons.

Such dynamic resolution information may exist for PO4 or above format program objects. The dynamic resolution information for a symbol consists of the symbol name, the CODE, CODE64, DATA, or DATA64 designation, and the name of the DLL from which the symbol is to be dynamically resolved. This is the same information as that provided on the IMPORT statement for the symbol. If this information is available via INCLUDE, the IMPORT control statement need not be input on a re-bind. If there is more than one entry for a particular symbol being imported, no message will be issued and the first occurrence will be retained.

-ALIASES
Specifies that the aliases of the input module be copied in and used as aliases for the output module. Aliases can be included only if:

- the input is a program object in either a UNIX file or a PDSE (and regardless of where the PDSE member name is)
- the input is a load module with the member name in the INCLUDE.

-NOATTR
Specifies that module attributes will not be copied from the input module.

-NOIMPORTS
Specifies that dynamic resolution information (if any) will not be copied from the input module.

-NOALIASES
Specifies that the aliases of the input will not be copied from the input module.

ddname
The name of a DD statement that describes a sequential data set, a PDS, a PDSE, or a z/OS UNIX file to be used as additional input to the binder. A DD statement must be supplied for every ddname specified in an INCLUDE statement. For a sequential data set, only ddname should be specified. For a PDS or PDSE without a member qualification in the JCL, at least one member name must also be specified. If only a single member is included, its member name can be specified in the JCL rather than on the control statement.

When the include source is a z/OS UNIX file, the DD statement must reflect the absolute or relative pathname of the file to be included. In this case, the *membername* is not applicable and should not be specified.

membername
The name of or an alias for a member of the library defined in the specified DD statement. The member name must not be specified on the DD statement.

pathname
The absolute or relative pathname for a z/OS UNIX file that can be up to 1023 bytes. Note that this is a direct specification for z/OS UNIX files. z/OS UNIX files can also be specified indirectly with a DD statement (see above). "Example 2" uses *pathname*. See "Binder syntax conventions" on page 97 for a discussion of continuations and lower case letters.

Placement: An INCLUDE statement can be placed before, between, or after object modules or other control statements.

Notes:

1. A NAME statement in any data set specified in an INCLUDE statement is invalid; the NAME statement is ignored. All other control statements are processed.
2. The INCLUDE statement is not allowed in a data set that is included from an automatic call library.
3. When invoking the binder using the TSO link command, an INCLUDE statement specifying a ddname of SYSLMOD will be allocated to the output library, unless SYSLMOD has been specifically allocated to another library.

Example 1

An INCLUDE statement can specify two data sets to be the input to the binder:

```
//OBJMOD     DD     DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//LOADMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=SHR
//SYSLIN     DD     *
  INCLUDE OBJMOD,LOADMOD(TESTMOD,READMOD)
   ⋮
/*
```

Two separate INCLUDE statements could also have been used in this example:

```
INCLUDE OBJMOD
INCLUDE LOADMOD(TESTMOD,READMOD)
```

Example 2

INCLUDE statements can reference both MVS data sets and z/OS UNIX files to be used as input to the binder. z/OS UNIX files can be specified directly on an INCLUDE statement, or indirectly through DD statements that in turn reference z/OS UNIX files:

```
//LOADMOD    DD     DSNAME=PROJECT.LOADLIB,DISP=SHR
//OBJECT2    DD     PATH='/sl/app1/pm3d3/dlla02',PATHDISP=(KEEP,KEEP)
//SYSLIN     DD     *
  INCLUDE LOADMOD(TESTMOD,READMOD)
  INCLUDE '/ml/app1/pm3d3/dlla01'
  INCLUDE OBJECT2
   ⋮
/*
```

INSERT statement

We do not recommend using the INSERT and OVERLAY statements for program objects. The binder supports the overlay format for compatibility only. If you use the OVERLAY statement, a program object will be created with a compatibility level of PM1 and, therefore, will not make use of the binder enhancements available in later releases. For more information on the use of the INSERT statement, see Appendix D, “Designing and specifying overlay programs,” on page 193.

The INSERT statement repositions a section from its position in the input sequence to a segment in an overlay structure. However, the sequence of sections within a segment is not necessarily the order of the INSERT statements.

If a symbol specified in the operand field of an INSERT statement is not present in the external symbol dictionary, it is entered as an external reference. If the reference has not been resolved at the end of primary input processing, the binder attempts to resolve it from the automatic call library. The syntax of the INSERT statement is:

INSERT	*sectionname*[,*sectionname*]...

sectionname
The name of the section to be repositioned. A particular section can appear only once within a program object or load module.

Placement: The INSERT statement must be placed in the input sequence following the OVERLAY statement that specifies the origin of the segment in which the section is positioned. If the section is positioned in the root segment, the INSERT statement must be placed before the first OVERLAY statement.

Notes:

1. Sections that are positioned in a segment must contain all address constants to be used during execution unless:
   - The A-type address constants are located in a segment in the path.
   - The V-type address constants used to pass control to another segment are located in the path. If an exclusive reference is made, the V-type address constant must be in a common segment.
   - The V-type address constants used with the SEGLD and SEGWT macro instructions are located in the segment.
2. Automatically called sections not specified on INSERT statements are added to the root segment.

Example

The following INSERT (and OVERLAY) statements specify the overlay structure shown in Figure 19 on page 113:

```
//          EXEC   PGM=IEWBLINK,PARM='OVLY,XREF,LIST'
            .
            .
            .
//SYSLIN     DD     *
  INSERT CSA
  INSERT CSB
  OVERLAY ALPHA
```

```
  INSERT CSC,CSD
  OVERLAY ALPHA
  INSERT CSE
/*
```

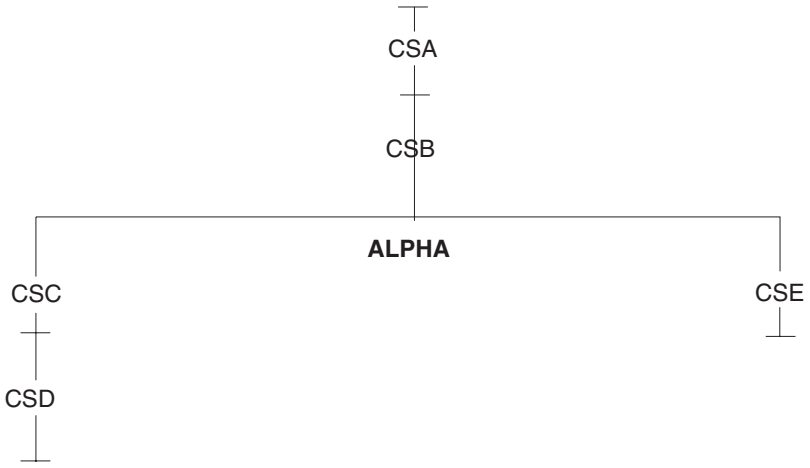


Figure 19. Overlay structure for INSERT statement example

LIBRARY statement

The LIBRARY statement can be used to specify:

- Additional automatic call libraries that contain modules used to resolve external references found in the program.
- Restricted no-call: External references that are not to be resolved by an automatic library call during the current binder job step.
- Never-call: External references that are not to be resolved by an automatic library call during this or any subsequent binder job step.

More than one library call specification can be made on the same LIBRARY statement.

The syntax of the LIBRARY statement is:

LIBRARY	{{*ddname*(*membername*[,...])} {(*externalreference*[,...])} {*(*externalreference*[,...])}},...

ddname
The name of a DD statement that defines a library from which the listed symbols will be included during automatic library call.

membername
The name of or an alias for a member of the specified library. Only those members specified are used to resolve references.

externalreference
An external reference that can be unresolved after primary input processing. The external reference is not to be resolved by automatic library call.

***** Indicates never-call; the external reference should never be resolved from an

automatic call library. If the * (asterisk) is missing, the reference is left unresolved during the current binder job step but can be resolved in a subsequent step.

If all binder input modules containing references to a specific symbol were bound with never-call, that symbol is not resolved by automatic library call during this binder run. However, if one or more input modules do not indicate a symbol as never-call, the binder attempts to resolve the symbol from the automatic call library.

Placement: A LIBRARY statement can be placed before, between, or after object modules or other control statements.

Notes:

1. If the unresolved external symbol is not a member or alias name in the library specified, the external reference remains unresolved unless defined in another input module.
2. If the NCAL option is specified, the LIBRARY statement has no effect.
3. Members included by automatic library call are placed in the root segment of an overlay program, unless they are repositioned with an INSERT statement.
4. The LIBRARY control statement is not processed immediately. If the same symbol appears on more than one LIBRARY statement, only the last occurrence is used.
5. Specifying an external reference for restricted no-call or never-call by means of the LIBRARY statement prevents the external reference from being resolved by automatic inclusion of the necessary module from an automatic call library; it does not prevent the external reference from being resolved if the module necessary to resolve the reference is specifically included or is included as part of an input module.
6. The LIBRARY statement is not allowed in a data set that is included from an automatic call library.

Example

This example shows all three uses of the LIBRARY statement:

```
//           EXEC   PGM=IEWBLINK,PARM='LET,XREF,LIST'
//TESTLIB    DD     DSNAME=PROJECT.TESTLIB,DISP=SHR
             .
             .
             .
//SYSLIN     DD     *
  LIBRARY TESTLIB(DATE,TIME),(FICACOMP),*(STATETAX)
/*
```

As a result, members DATE and TIME from the additional library TESTLIB are used to resolve external references. FICACOMP and STATETAX are not resolved; however, because the references remain unresolved, the LET option must be specified on the EXEC statement if the module is to be marked executable. In addition, STATETAX will not be resolved in any subsequent reprocessing by the binder.

MODE statement

The MODE statement specifies the addressing mode for all the entry points into the program module (the main entry point, its true aliases, and all the alternate entry points) and the residence mode for the program module. The syntax of the MODE statement is:

MODE	*modespec*[,*modespec*]

modespec

One or both of the following:

- The designation of an addressing mode for the output program object or load module by one of the following:

  AMODE(24)
  AMODE(31)
  AMODE(64)
  AMODE(ANY)
  AMODE(MIN)

  Specifying AMODE(MIN) causes the most restrictive AMODE of all control sections within the program module to be assigned.

  See “AMODE: Addressing mode option” on page 73 for more information about specifying AMODE.

- The designation of residence mode for the output program object or load module by one of the following:

  RMODE(24)
  RMODE(ANY)
  RMODE(SPLIT)

  See “RMODE: Residence mode option” on page 89 for more information about specifying RMODE.

Placement: The MODE control statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module, if one is present.

Notes:

1. If more than one MODE control statement is encountered in the binding of a program object or load module, the last valid AMODE and RMODE specifications are used.
2. The binder treats AMODE and RMODE values independently until they are required for output processing. At this time the combination of AMODE and RMODE values for each entry point are checked for conflict. See “AMODE and RMODE combinations” on page 27 for information on AMODE and RMODE compatibility.
3. The addressing mode assigned by the MODE control statement overrides the separate addressing modes found in the ESD data for the control sections within which the entry points are located. The addressing mode assigned by the MODE control statement overrides the addressing mode assigned by the AMODE parameter in the PARM field of the EXEC statement. A specified AMODE value applies to all entry points in the module, and is stored in all generated directory entries.
4. The residence mode assigned by the MODE control statement overrides the residence mode accumulated from the input control sections and private code. The residence mode assigned by the MODE control statement also overrides the residence mode assigned by the RMODE parameter in the PARM field of the EXEC statement. A specified RMODE value applies to the entire module, unless the SCTR (scatter) option has been specified, and is stored in all generated directory entries.

Example

In this example, an output module, NEWMOD, is created. It is given an alias of TESTMOD, the residence mode for the module is ANY, and the addressing mode for both the main entry point, NEWMOD, and the true alias, TESTMOD, is 31. The addressing and residence modes allow the program to be loaded into 31-bit addressable virtual storage.

```
//SYSLMOD  DD  DSN=USER.TESTPROG,DISP=OLD
//SYSLIN   DD  *
           .
           .
           .
    MODE   AMODE(31),RMODE(ANY)
    ALIAS  TESTMOD
    NAME   NEWMOD
/*
```

NAME statement

The NAME statement specifies the name of the program module created from the preceding input modules, and serves as a delimiter for input to the program module. As a delimiter, the NAME statement allows you to create more than one program module in one binder step. The NAME statement can also indicate that the module replaces an identically named module in the output program library. The syntax of the NAME statement is:

NAME	*membername*[(R)]

membername
The name to be assigned to the program object or load module created from the preceding input modules.

(R)
Indicates that this program module replaces an identically named module in the output module library, and that any aliases specified on ALIAS statements replace identically named aliases. If the module is not a replacement, (R) is ignored.

Placement: The NAME statement is placed after the last input module or control statement to be used for the output module.

Notes:

1. Any ALIAS statement must precede the NAME statement.
2. If you are binding a program object, only the aliases specified on ALIAS statements are kept for the program object. Any other aliases for the replaced program object are deleted from the directory of the program library. If you are binding load modules, any aliases for the replaced load modules that are not themselves replaced are kept and point to the old load module.
3. If a name is not specified either on the NAME statement or on the DD statement for the SYSLMOD data set, and the SYSLMOD data set is a PDS or PDSE, the binder will assign the name TEMPNAM*n*, using values 0-9 for *n*. The binder will not save the module if the names TEMPNAM0 through TEMPNAM9 are already in use. This assignment of temporary names does not take place if the SYSLMOD data set is a z/OS UNIX file. Instead, the binder issues an error message stating its inability to save the output module.

4. If the (R) value is not specified, and a member of the same name already exists in the output module library, the binder will not replace the module or save it under another name.
5. Normally, the binder does not replace an executable module with a nonexecutable module even if the (R) value is specified. You can specify the STORENX option to override this default action. See “STORENX: Store not-executable module” on page 91 for a further description.
6. A NAME statement found in a data set other than the primary input data set is invalid. The statement is ignored.
7. The IEWBLDGO binder entry point does not accept a NAME statement.
8. If you do not specify the (R) parameter when processing a z/OS UNIX file, the binder issues an informational message.

Example

In this example, two output modules, RDMOD and WRTMOD, are produced by the binder in one job step:

```
//SYSLMOD    DD     DSNAME=PROJECT.AUXMODS,DISP=SHR
//NEWMOD     DD     DSNAME=&&WRTMOD,DISP=OLD
//SYSLIN     DD     DSNAME=&&RDMOD,DISP=OLD
//            DD     *
  NAME RDMOD(R)
  INCLUDE NEWMOD
  NAME WRTMOD(R)
/*
```

The first time modules RDMOD and WRTMOD are created in the module library AUXMODS, the (R) option is ignored. When the same modules are rebound using the same control statements, the (R) option results in a replacement of the old modules.

ORDER statement

The ORDER statement indicates the sequence in which control sections or named common areas appear in the output program object or load module. The control sections or named common areas appear in the sequence they are specified on the ORDER statement. The syntax of the ORDER statement is:

ORDER	*sectionname*[(P)]

section name
The name of the section to be sequenced.

(P)
Indicates the starting address of the control section or named common area is on a page boundary within the program object or load module. The control sections or common areas are aligned on 4KB page boundaries, unless the ALIGN2 option has been specified.

Placement: An ORDER statement can usually be placed before, between, or after object modules or other control statements.

Notes:

1. When multiple ORDER statements are used, their sequence further determines the sequence of the control sections or named common areas in the output module. If the same common area or control section is listed on more than one ORDER statement, the binder uses the sequence stated on the last request.

2. The control sections and common areas named as operands can appear in either the primary input or the automatic call library, or both.
3. If a control section or a named common area is changed by a CHANGE or REPLACE control statement and sequencing is desired, specify the new name on the ORDER statement.

Example

In the statements shown in Figure 20, the control sections in the module LDMOD are arranged by the binder according to the sequence specified on ORDER statements. The page boundary alignments and the control section sequence made as a result of these statements are shown in Figure 20. Assume each control section is less than 1KB in length.

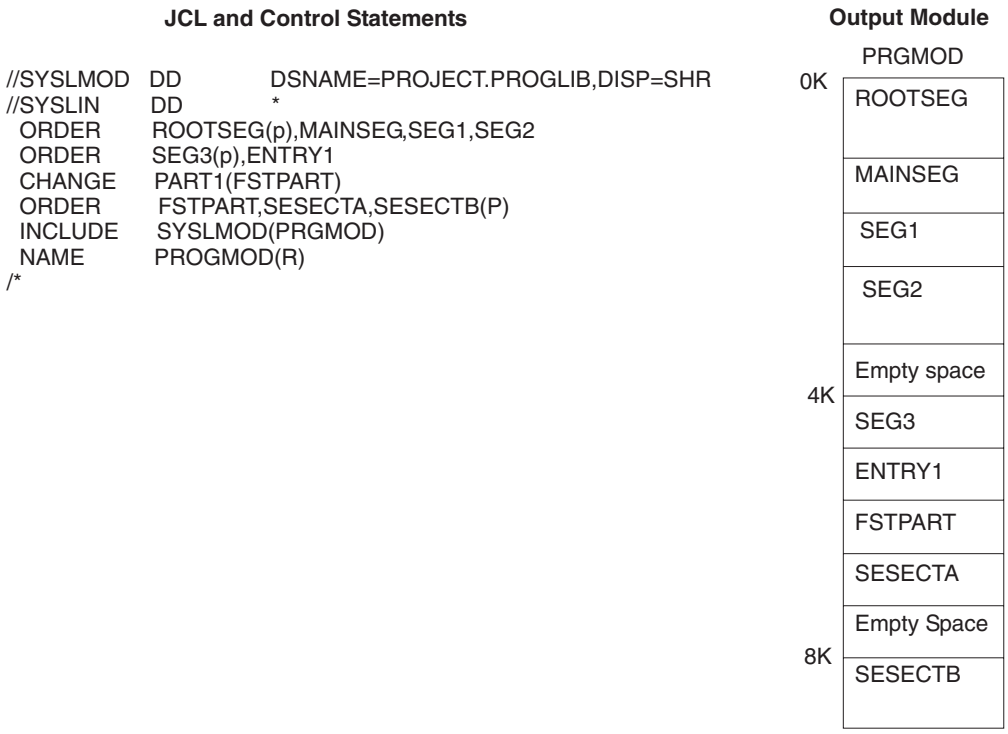


Figure 20. Example of an output module for the ORDER statement. The control section name PART1 is changed by a CHANGE statement to FSTPART. The ORDER statement refers to the control section by its new name.

OVERLAY statement

We do not recommend using the INSERT and OVERLAY statements for program objects. The binder supports the overlay format for compatibility only. For more information on the use of the OVERLAY statement, see Appendix D, “Designing and specifying overlay programs,” on page 193.

The OVERLAY statement indicates the beginning of an overlay segment and, optionally, also of an overlay region. Because a segment or a region is not named, you identify it by giving its origin (or load point) a symbolic name. This name is then used on an OVERLAY statement to signify the start of a new segment beginning at that origin. The syntax of the OVERLAY statement is:

OVERLAY	*symbol*[(REGION)]

symbol
The symbolic name assigned to the origin of a segment. This symbol is not related to external symbols in the module.

(REGION)
Specifies the origin of a new region, as well as a segment.

Placement: The OVERLAY statement must precede the first module of the next segment, the INCLUDE statement specifying the first module of the segment, or the INSERT statement specifying the control sections to be positioned in the segment.

Notes:

1. The OVLY option must be specified on the EXEC statement when OVERLAY statements are to be used.
2. The sequence of OVERLAY statements should reflect the order of the segments in the overlay structure from top to bottom, left to right, and region by region.
3. No OVERLAY statement should precede the root segment.

Example

The following OVERLAY and INSERT statements specify the overlay structure in Figure 21 on page 120.

```
//          EXEC   PGM=IEWBLINK,PARM='OVLY,XREF,LIST'
  ⋮
//SYSLIN    DD     DSNAME=&&OBJ,...
//          DD     *
  INSERT CSA
  OVERLAY ONE
  INSERT CSB
  OVERLAY TWO
  INSERT CSC
  OVERLAY TWO
  INSERT CSD
  OVERLAY ONE
  INSERT CSE,CSF
  OVERLAY THREE(REGION)
  INSERT CSH
  OVERLAY THREE
  INSERT CSI
/*
```

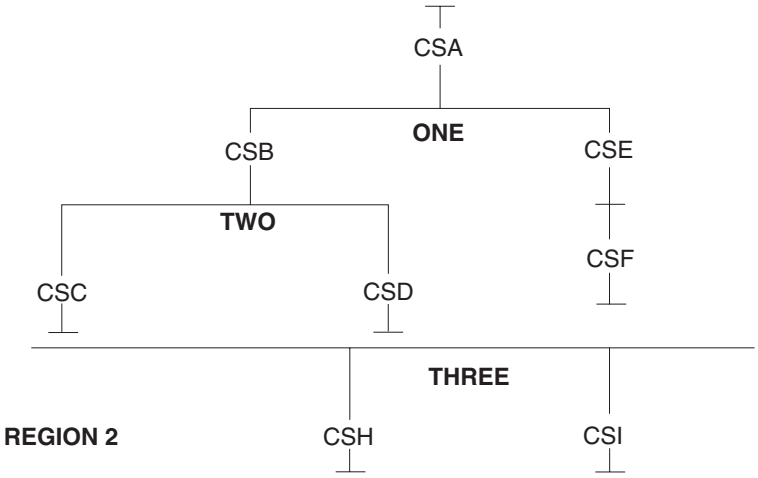


Figure 21. Example of an overlay structure for the OVERLAY statement

PAGE statement

The PAGE statement aligns a control section or named common area on a 4KB page boundary in the program object or load module. The syntax of the PAGE statement is:

PAGE	*sectionname...*

section name
The name of the section to be aligned on a page boundary.

Placement: The PAGE statement can be placed before, between, or after object modules or other control statements.

Notes:

1. If a section is changed by a CHANGE or REPLACE control statement, and page alignment is wanted, specify the new name in the PAGE statement.
2. The sections named can appear in either the primary input or the automatic call library, or both.
3. If the ALIGN2 option has been specified, sections listed on the PAGE statement will be aligned on 2KB boundaries.

Example

In this example, the sections in the module PRGMOD are aligned on page boundaries as specified in the following PAGE statement:

```
PAGE ALIGN,BNDRY4K,EIGHTK
```

The job control statements and binder control statements as well as the output program object or load module are shown in Figure 22 on page 121. Assume each control section is 3KB in length.

JCL and Control Statements

```
//LKED      EXEC    PGM=IEWBLINK,PARM=,...
//SYSLMOD   DD      DSNAME=PROJECT.LOADLIB,DISP=SHR
//SYSLIN    DD      *
  PAGE      ALIGN,BNDRY4K,EIGHTK
  INCLUDE   SYSLMOD(PRGMOD)
  NAME      PROGMOD(R)
/*
```

Output Module

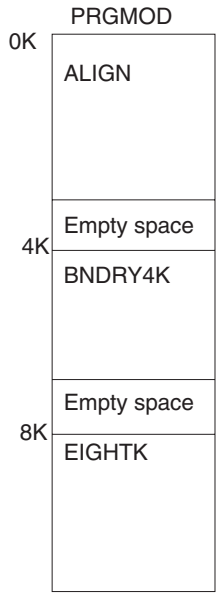


Figure 22. Example of an output module for the PAGE statement

RENAME statement

The RENAME statement allows for the renaming of specific symbols. An old symbol name can be renamed to a new symbol name that can then be used to resolve references when binding a module. The rename requests take place only after the binder attempts to resolve the original names. The new names are then used during the binder's final autocall in order to resolve any references previously unresolved.

The syntax of the RENAME statement is:

RENAME	*oldname,newname*

oldname
The symbol to be renamed. Its maximum length is 32767 bytes.

newname
The symbol name to which the *oldname* should be changed. Its maximum length is 32767 bytes.

To continue either of the symbols on multiple lines, code a nonblank character in column 72 of each line. Columns 73 to 80 of each line are ignored.

Placement: The RENAME statement can be placed before, between, or after object modules or other control statements. They do not take effect, however, until an AUTOCALL statement is processed, or, in the absence of AUTOCALL statements, until after the binder's final autocall processing takes place.

Notes:

1. The only immediate result of the RENAME control statement is that the rename request is added to the binder's list of such requests. RENAME processing takes place only after all possible references have been resolved with the names as they were specified on input.

2. This statement is the functional equivalent of the prelinker's RENAME control statement. It should be noted, however, that the *SEARCH* parameter of the prelinker's statement is not supported by the binder.
3. RENAME will only affect symbols that are marked as *renameable*. Since traditional object modules and load modules do not support the renameable attribute, RENAME will have no effect on symbols originating from modules in those formats. The renameable attribute is supported by GOFF, and it is also set for XSD records with the "mapped" bit off (from XOBJ modules).
4. RENAME will have no effect on symbols originating from PR records (pseudoregister or part references).
5. RENAME will have no effect on imported symbols.

Example

```
  ⋮
//TAXES      DD     PATH='/sl/finance/app1/dlltxs',PATHDISP=(KEEP,KEEP)
//SYSLIB     DD     DSNAME=PROJECT.OBJLIB,DISP=SHR
//SYSLIN     DD     *
  INCLUDE  TAXES
  RENAME   Compute_98_Taxes_Schedule2,Taxes98
  ⋮
/*
```

REPLACE statement

The REPLACE statement is used to replace or delete external symbols. The external symbol can name a section, an entry point, an external reference, or a pseudoregister.

One section can be replaced with another. All references within the input module to the old section are changed to the new section. Any external references to the old section from other modules are unresolved unless changed.

A section can be deleted. The section name is deleted from the external symbol dictionary. External references from other modules to a deleted section also remain unresolved. If there are references to any address within a deleted section, the section name in changed to an external reference.

If the first symbol in the REPLACE statement refers to a symbol that is not a section or common area, the results will be the same as if a CHANGE statement were coded. The first symbol is replaced by the second symbol. The first symbol is deleted when the second symbol is omitted.

The syntax of the REPLACE statement is:

REPLACE	*externalsymbol1*[(*externalsymbol2*)]...

externalsymbol1, externalsymbol2
Names an external symbol to be replaced or deleted. If you only specify *externalsymbol1*, the external symbol is deleted. If you specify *externalsymbol2* in parentheses following *externalsymbol1*, *externalsymbol1* is replaced by *externalsymbol2*. You can delete or replace any number of external symbols with one REPLACE statement.

Placement: The REPLACE statement must immediately precede either the module containing the external symbol to be replaced or deleted, or the INCLUDE

statement specifying the module. The scope of the REPLACE statement is across the immediately following program or object module.

Notes:

1. If during automatic library call the replacement symbol is still undefined in the module, the binder attempts to resolve the reference from SYSLIB.
2. When a section containing unresolved external references is deleted, the binder removes these references from the ESDs.
3. When using the binder, if no INCLUDE statement follows the REPLACE statement, the request is ignored.
4. If the REPLACE statement appears in a module included from a data set in an automatic call library, it will be ignored if it is not followed by a module from the same data set.
5. Restrictions apply whenever both CHANGE and REPLACE operations are performed on the same included program or object module. You might need to delete one of several sections and at the same time rename references to that section (all within the scope of the same INCLUDE) to some other external symbol. To change more than one entry name within the original section to a single new external symbol, you must specifically include the section that resolves the new external symbol, prior to the change operation.
6. When using a REPLACE statement to replace or delete a named common area, the common area must be defined in the first program or object module following the REPLACE statement.
7. When deleting an entry name, if there are any references to it within the same input module, the entry name is changed to an external reference.

Example

In this example, assume that section INT7 is in member LOANCOMP and that section INT8, which is to replace INT7, is in data set &&NEWINT. Also assume that section PRIME in member LOANCOMP is deleted.

```
//NEWMOD     DD     DSNAME=&&NEWINT,DISP=(OLD,DELETE)
//OLDMOD     DD     DSNAME=PROJECT.PROGLIB,DISP=SHR
//SYSLIN     DD     *
  ENTRY MAINENT
  INCLUDE NEWMOD
  REPLACE INT7(INT8),PRIME
  INCLUDE OLDMOD(LOANCOMP)
  NAME LOANCOMP(R)
/*
```

As a result, INT7 is removed from the input module described by the OLDMOD DD statement, and INT8 replaces INT7. All references to INT7 in the input module now refer to INT8. Any references to INT7 from other modules remain unresolved. If there are no references to PRIME in LOANCOMP, section PRIME is deleted; the section name is also deleted from the external symbol dictionary.

SETCODE statement

The SETCODE statement assigns a specified authorization code to the output load module or program object. The authorization code is placed in the directory entry for the output load module or program object.

The binder allows any numeric value between 0 and 255. The MVS Authorized Program Facility (APF) determines that a module is authorized if the authorization code has a value of 1. The module is unauthorized if the authorization code has

any other value. Refer to *z/OS MVS Programming: Authorized Assembler Services Guide* for additional information on the APF.

The syntax of the SETCODE statement is:

SETCODE	AC(*authorizationcode*)

authorizationcode
A decimal number from 0 to 255. Specifying AC() results in an authorization code of zero.

Placement: A SETCODE statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module, if one is present.

Notes:

1. The authorization code assigned by the SETCODE statement overrides the authorization code assigned by the AC parameter in the PARM field of the EXEC statement.
2. If more than one SETCODE statement is encountered in the bind of a load module or program object, the last valid authorization code assigned is used.

Example

In this example, an authorization code of 1 is assigned to the output module MOD1.

```
//LKED       EXEC   PGM=IEWBLINK
//SYSPRINT   DD     SYSOUT=A
//SYSLMOD    DD     DSNAME=SYS2.LINKLIB,DISP=OLD
//SYSLIN     DD     DSNAME=&&LOADSET,DISP=(OLD,PASS)
//           DD     *
  SETCODE           AC(1)
  NAME              MOD1(R)
/*
```

SETOPT statement

The SETOPT statement allows you to set options at the module level, rather than the job step level as in the binder batch parameter string. This allows you to set module attributes when a number of modules are being bound separately in a single MVS job step.

SETOPT accepts a string of parameter specifications as if it had been entered on the PARM parameter of the EXEC JCL statement. The options you specify are valid only until after the next NAME control statement is processed or until an end-of-file condition is detected in SYSLIN.

The syntax of the SETOPT statement is:

SETOPT	PARM(*parm*)

PARM(*parm*)
Accepts a string of parameter specifications as if it had been entered on the PARM parameter of the EXEC JCL statement. It follows the same syntax rules as the binder batch execution parameter string. The following batch options cannot be set using the SETOPT control statement:

- COMPAT
- EXITS

- LINECT
- MSGLEVEL
- OPTIONS
- PRINT
- SIZE
- TERM
- TRAP
- WKSPACE

In addition, the single keyword form of REUS cannot be used with SETOPT.

See Chapter 6, “Binder options reference,” on page 67 for more information on the options that can be specified in the PARM field of the EXEC statement.

Note: The SETOPT control statement is not recommended for use with SMP/E. Options specified on SETOPT are not processed by SMP/E and can cause conflicts during installation.

SETSSI statement

The SETSSI statement specifies hexadecimal information to be placed in the system status index of the directory entry for the output module. The syntax of the SETSSI statement is:

SETSSI	*ssi-info*

ssi-info
Represents 8 hexadecimal characters (0 through 9 and A through F) to be placed in the 4-byte system status index of the output module library directory entry.

Placement: The SETSSI statement can be placed before, between, or after object modules or other control statements. If one is present, it must precede the NAME statement for the module.

Notes:

1. The SETSSI statement overrides any SSI option included in the PARM field of the EXEC statement.
2. A SETSSI statement should be provided whenever an IBM-supplied program module is reprocessed by the binder. If the statement is omitted, no system status index information is present.

Chapter 8. Interpreting binder listings

This appendix contains an overview of the binder output. This output is written to SYSPRINT, SYSLOUT, or another ddname assigned to the PRINT file (using the FILES parameter) on the STARTDialog call. Except where noted, all outputs apply to both batch entry points (IEWBLINK and IEWBLDGO) and to both load modules and program objects.

Linkage editor and batch loader outputs are described in “Interpreting linkage editor output” on page 167 and “Interpreting batch loader output” on page 173.

The output data is divided into a number of categories, some that always appear in the output listing and others that appear depending on the options selected. The categories are:

- Header
- Input Event Log
- Private Section List
- Program Module Map
- Renamed Symbol Table
- Cross-Reference Table
- Imported and Exported Symbol Table
- Operation Summary
- Long-symbol Cross-Reference Table
- Short Mangled Name Report
- Abbreviation/Demangled Name Report
- DDname to pathname cross reference
- Message Summary Report

Header

The header is written at the beginning each section of the output. The header contains information on the release and modification level and on how the binder was invoked.

- Name, version, release, and modification level of the binder
- Time, day, and date of invocation
- Job name, step name, program name, and (if one has been used) procedure name when invoked by use of a batch interface. When invoked via the application programming interface, the binder prints the contents of the CALLERID field from the STARTD call.
- Binder entry point name.

Input event log

The input event log is a chronological log of the events that took place during the input phase of binder operation. Its presence is controlled by the LIST option. If LIST(OFF) or NOLIST is specified, no input event log is generated. If LIST(STMT), LIST, or LIST(SUMMARY) is specified, only input events pertaining to control statements are logged. If LIST(NOIMP) is specified, messages pertaining to the import control statement are suppressed, while those generated by other control statements and binder calls continue to be logged. When processing DLLs that contain a large number of IMPORT control statements in their side files, this option helps to reduce the number of messages logged while still providing information

about other binder processing. If LIST(ALL) is specified, all input events are logged (such as those initiated by binder function calls as well as those initiated by control statements).

Figure 23 contains a sample input event log. The log can include:

- The list of processing options used in the binder invocation.
- Errors with the invocation parameter (binder or batch loader options)
- Line by line summary of functions performed during the input phase. Each bind operation is treated separately: a control statement is printed, followed by a summary of the function performed and the complete names of the objects operated upon.
- Errors encountered during the input phase.

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(TESTHIGH) STEP(BIND1   ) PGM= IEWL
IEW2278I B352 INVOCATION PARAMETERS - LIST(ALL),MAP,XREF,NCAL
IEW2322I 1220  1    INCLUDE MYLIB(PROGBCAD)
IEW2308I 1112 SECTION PROGBCAD HAS BEEN MERGED.
IEW2322I 1220  2    INCLUDE MYLIB(PROGBCDS)
IEW2308I 1112 SECTION PROGBCDS HAS BEEN MERGED.
IEW2322I 1220  3    INCLUDE MYLIB(PROGBCOV)
IEW2308I 1112 SECTION PROGBCOV HAS BEEN MERGED.
IEW2322I 1220  4    MODE AMODE(31),RMODE(ANY)
IEW2322I 1220  5    ENTRY PROGDCTL
IEW2322I 1220  6    ALIAS PROGIND
IEW2322I 1220  7    ALIAS PROGSTAK
IEW2322I 1220  8    NAME PROGIND0(R)
IEW2454W 9203 SYMBOL PROGXCLW UNRESOLVED.  NO AUTOCALL (NCAL) SPECIFIED.
IEW2454W 9203 SYMBOL PROGXCWL UNRESOLVED.  NO AUTOCALL (NCAL) SPECIFIED.
```

Figure 23. Sample binder input event log

Note: In the binder, message IEW2308I replaces the previous message IEW2307I.

Private section list

Figure 24 contains a sample private section list report.

This report will appear immediately before the module map if LISTPRIV=YES was specified as a binder option and if there are any unnamed sections.

```
*** U N N A M E D  S E C T I O N ***

              ------- SOURCE -------
 NAME         DDNAME   SEQ   MEMBER


 $PRIV000010   LIB1     01   BTEST10A

*** E N D   O F  U N N A M E D  S E C T I O N S ***
```

Figure 24. Sample binder private section list report

Program module map

A map of the program module is generated if the MAP option was specified for the run. Figure 25 on page 131 and Figure 26 on page 133 contain sample program module maps (one for a simple module and one for an overlay module). Each text class is mapped showing each section or external label on a separate line and INCLUDING information about the source of the section. A “SOURCE” column indicates the data set (by ddname and concatenation sequence number) and member from which each section was included. Map entries are sequenced by module location within class or overlay segment.

The following describes the detailed line information included in the module map:

- SECTION OFFSET - The location of the section or label relative to the start of the element (class section) in which it is defined. Section offset is printed only for labels, not sections.
- CLASS OFFSET/MODULE OFFSET - The location of the section or label relative to the start of the class or overlay segment.
- NAME - The name of the entity being mapped. An asterisk preceding the name indicates that the section was included during the autocall phase.

  Some section types do not have external names and are displayed as follows:
  - $SEGTAB - Overlay segment table
  - $ENTAB - Overlay segment entry table
  - $PRIVxxxxxx - Private code where *xxxxxx* is a unique hexadecimal value starting at X'000001'
  - $BLANKCOM - Blank (unnamed) common
- TYPE - The label type of the entity being mapped:
  - CSECT - Control section
  - LABEL - External label
  - COMMON - Named or unnamed common
  - SEGTAB - Overlay segment table
  - ENTAB - Overlay segment entry table
- LENGTH - The length in hexadecimal bytes of the section or segment. If TYPE is LABEL, this field is blank.
- SOURCE - The ddname, concatenation sequence number, and optionally the member name from which this section is included.

When reporting the source of a Section brought in from an archive file, the Binder Map will be changed to list the name of the archive file member from which the Section was included (in the column headed MEMBER). For symbols resolved via the C370LIB directories, the member name listed will be the PDS/PDSE member name, not the name of the symbol.

The last item in the module map is usually the data set summary. It contains one entry for each combination of ddname and concatenation sequence number referenced in the module map and displays the corresponding data set name. These 8-byte pseudo ddnames are used in the module map and other reports in order to improve the reports’ readability. The data set summary cross-references the pseudo ddnames to their corresponding z/OS UNIX file names. A pseudo ddname is of the form '/000000n', where 'n' is a number that increases as new z/OS UNIX files are processed by the binder.

Data sets and libraries from which no members were included do not appear in the data set summary.

If any symbols appear as references in the symbol table (ESD) of one or more input modules, but are not the target of any references in the code, an unreferenced symbol table will be printed. The symbols in this table will not appear in the cross-reference listing but if they are unresolved, may result in error messages being issued by the binder.

Simple module

The following figure illustrates a simple module, containing one text class (B_TEXT) and the pseudoregister vector (B_PRV). Each text class begins with a class header containing the class name, its length in bytes, and significant bind and load attributes of the class:

- CAT indicates that the class is a concatenation of all participating sections.
- LOAD indicates that the class will be loaded when the module is loaded.
- RMODE=ANY indicates that this class can be placed above the 16 MB line.

All CAT-type text classes consist of sections (CSECTs) and labels.

The second class, B_PRV, represents the pseudoregister vector (PRV), if one is present. It replaces the special PRV display that appeared in earlier releases of the binder. Its attributes are:

- MRG indicates that the class consists of parts, which are merged by part name.
- NOLOAD means that the class will not be loaded with the module.

There are several differences between the MRG and CAT classes. Since all pseudoregisters are located in the same section, section offset and class offset are identical; only one is printed. The entity is PART rather than CSECT or LABEL, each part representing a single pseudoregister or external data item. Finally, SOURCE is not displayed, since all parts are created by the binder.

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(PMSBC321) STEP(BIND2   ) PGM= IEWBLINK
IEW2278I B352 INVOCATION PARAMETERS - LIST(ALL),NCAL,LET,OVLY,MAP,XCAL

                        *** M O D U L E  M A P ***

 ---------------
 CLASS  B_TEXT            LENGTH =    A20  ATTRIBUTES=CAT,    LOAD, RMODE=ANY
                          OFFSET =      0 IN SEGMENT 001      ALIGN = DBLWORD
 ---------------

  SECTION    CLASS                                                 ------- SOURCE --------
   OFFSET   OFFSET  NAME                 TYPE    LENGTH  DDNAME   SEQ  MEMBER

                0  CEESTART            CSECT        7C  OBJ       01  C955A03

               80  EDCINPL           *  CSECT        28  /0000001  01  EDCINPL

               A8  STRCMP            *  CSECT        10  SYSLIB    02  STRCMP

               B8  PRINTF            *  CSECT        10  SYSLIB    02  PRINTF

               C8  EDC@@1FC          *  CSECT        10  SYSLIB    06  exit
        0      C8     exit               LABEL
        0      C8     EDC#EXIT           LABEL

               D8  CEESG003          *  CSECT       128  SYSLIB    06  exit

              200  puts               *  CSECT        10  SYSLIB    06  puts

              210  printf             *  CSECT        10  SYSLIB    06  printf

              220  CEEROOTA           *  CSECT       1F8  SYSLIB    02  CEEROOTA

              418  CEEBETBL           *  CSECT        28  SYSLIB    01  CEEBETBL

              440  CEEBPUBT           *  CSECT        70  SYSLIB    02  CEEBPUBT

              4B0  CEEBTRM            *  CSECT        B0  SYSLIB    02  CEEBTRM

              560  CEEBLLST           *  CSECT        60  SYSLIB    02  CEEBLLST
       10     570     CEELLIST           LABEL

              5C0  CEEBINT            *  CSECT         8  SYSLIB    02  CEEBINT

              5C8  CEEBPIRA           *  CSECT       280  SYSLIB    02  CEEINT
        0     5C8     CEEINT             LABEL
        0     5C8     CEEBPIRB           LABEL
        0     5C8     CEEBPIRC           LABEL

              848  CEECPYRT           *  CSECT        F0  SYSLIB    02  CEEINT

              938  CEEARLU            *  CSECT        B8  SYSLIB    02  CEEARLU

              9F0  CEETGTFN           *  CSECT        10  SYSLIB    01  CEETGTFN

              A00  CEETLOC            *  CSECT        20  SYSLIB    01  CEETLOC
```

Figure 25. Sample binder module map (Part 1 of 2)

```
-------------
CLASS B_PRV            LENGTH=      18 ATTRIBUTES=MRG,NOLOAD
-------------

      CLASS
     OFFSET  NAME               TYPE    LENGTH       SECTION

          0  GFLGA              PART           1
          1  GFLGE              PART           1
          2  GFLGC              PART           1
          3  GFLGC              PART           1
          4  COUNTF             PART           2
          8  MASTER             PART           4
         10  B_TOKEN            PART           8

           ***  DATA SET SUMMARY  ***

 DDNAME    CONCAT   FILE IDENTIFICATION
 OBJ         01     DFPFT.WORKLIB.OBJECT
 /0000001    01     /DFPFT/APP1/EDCINPL
 SYSLIB      01     DFPFT.WORKLIB.POSIX.RTL.UT2.SCEELKED
 SYSLIB      02     DFPFT.WORKLIB.CEE.V1R7M0.SCEELKED
 SYSLIB      06     A860059.SCEELKED.LONGNAME

           ***  SYMBOL REFERENCE NOT ASSOCIATED WITH ANY ADCON ***

TYPE    SCOPE   NAME
 ER       M     WEAK
 ER       L     DANGLER

           *** E N D  O F  M O D U L E  M A P ***
```

Figure 25. Sample binder module map (Part 2 of 2)

Figure 26 on page 133 shows an overlay format module map, containing three overlay segments and a pseudoregister vector. Note that all text is contained in class B_TEXT, a requirement of overlay programs.

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(TESTHIGH) STEP(BIND2   ) PGM= IEWBLINK
IEW2278I B352 INVOCATION PARAMETERS - LIST(ALL),NCAL,LET,OVLY,MAP,XCAL


                        ***  M O D U L E   M A P  ***

-------------
CLASS: B_TEXT
-------------
                        LENGTH=   11848 ATTRIBUTES = CAT,  LOAD, RMODE 24
                        OFFSET=       0 IN SEGMENT 001     ALIGN = DBLWORD


SEGMENT 001 REGION 001 LENGTH:    A370 ATTRIBUTES: OVERLAY     RMODE: 24

 SECTION    MODULE                                          ------- SOURCE ------
  OFFSET    OFFSET   NAME                TYPE     LENGTH   DDNAME  SEQ  MEMBER

               0   PROGBCAD           CSECT       1868   MYLIB     1  PROGBCAD

            1868   PROGBCDS           CSECT       13E8   MYLIB     1  PROGBCDS

            2C50   PROGBCOV           CSECT        190   MYLIB     1  PROGBCOV

            2DE0   PROGBIND           CSECT        C30   SYSLIB    1  PROGBIND

            3A10   PROGBRAC           CSECT       15D0   SYSLIB    2  PROGBRAC

            83F8   PROGBUPA           CSECT       1A20   SYSLIB    2  PROGBUPA

            9E18   PROGPMMB           CSECT        528   SYSLIB    2  PROGPMMB
     424    A23C      PROGPARB           LABEL

            A340   PROGCDEF           CSECT        3F0   SYSLIB    1  PROGCDEF

SEGMENT 002 REGION 001 LENGTH:    32E0 ATTRIBUTES: OVERLAY     RMODE: 24

 SECTION    MODULE                                          ------- SOURCE ------
  OFFSET    OFFSET   NAME                TYPE     LENGTH   DDNAME  SEQ  MEMBER

               0   PROGMX21           CSECT        868   SYSLIB    1  PROGBCAD

             868   PROGGROV           CSECT        3E8   SYSLIB    1  PROGBCDS

             C50   PROGWYY            CSECT        490   MYLIB     1  PROGBCOV

            10E0   PROGR2D2           CSECT        C30   MYLIB     1  PROGBIND

            1D10   PROGC3PO           CSECT       15D0   MYLIB     1  PROGBRAC
```

Figure 26. Sample binder module map - Overlay (Part 1 of 2)

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(TESTHIGH) STEP(BIND2   ) PGM= IEWBLINK

SEGMENT 003 REGION 001 LENGTH:    3E38 ATTRIBUTES: OVERLAY     RMODE: 24

 SECTION    MODULE                                          ------- SOURCE ------
  OFFSET    OFFSET   NAME                TYPE     LENGTH   DDNAME  SEQ  MEMBER

               0   OBI_WAN            CSECT        720   MYLIB     1  PROGBCAD

             720   JABBA              CSECT        9A0   MYLIB     1  PROGBCDS

            10C0   STARWARS           CSECT        440   MYLIB     1  PROGBCOV
       0    10C0      LUKE               LABEL
       4    10C4      LEAH               LABEL
       8    10C8      DARTH              LABEL

            1500   YODA               CSECT       2030   MYLIB     1  PROGBIND

            3530   CHEWBACA           CSECT        904   MYLIB     1  PROGBRAC


-------------
CLASS: B_PRV           LENGTH:     D70 ATTRIBUTES: MRG, NOLOAD
-------------

           CLASS
           OFFSET   NAME                TYPE     LENGTH

               0   INFILE             PART           4
               4   OUTPUT1            PART           4
               8   WORK1              PART         D56
             D60   SYSPRINT           PART           4
             D68   MESSAGEH           PART           8

         ***  DATA SET SUMMARY  ***

 DDNAME    CONCAT    FILE IDENTIFIER

 MYLIB        1      JONES.PROJECT6.LOADLIB
 SYSLIB       1      DEPT77.OBJLIB
              2      DEPT83.OBJLIB

         ***   E N D   O F   M O D U L E   M A P  ***
```

Figure 26. Sample binder module map - Overlay (Part 2 of 2)

Renamed-symbol cross-reference table

The renamed-symbol cross-reference table is printed only if one or more names were renamed for symbol resolution purposes. The table shows the correspondence between the new (renamed) and the source symbols.

The binder normally processes symbols exactly as received from the compiler. However, certain symbolic references generated by the C/C++ and other compilers can be renamed by the binder if they contain long or mixed-case names (″L-names″) and cannot be resolved by the L-name during autocall. During renaming, the L-name reference is replaced by its equivalent short name. Such replacements, whether resolved or not, will appear in the Renamed-Symbol Table.

Figure 27 depicts three renamed symbols, the last of which is differentiated as a part or pseudoregister name.

```
***  RENAMED SYMBOL CROSS REFERENCE  ***
--------------------
RENAMED SYMBOL
      SOURCE SYMBOL
--------------------

 function9_40__FPfPi
       function9_xxxxxxxxx20xxxxxxxx30xxxxxxxx4__FPfPi

 function2_31__sqrt
       function2_xxxxxxxxx20xxxxxxxx3__sqrt

+function7_41__FPfPi
       function7_xxxxxxxxx20xxxxxxxx30xxxxxxxx4__FPfPi

+ = PART OR PSEUDO REG
 ***  END OF RENAMED SYMBOL CROSS REFERENCE  ***
```

Figure 27. Sample binder renamed-symbol cross-reference

Cross-reference table

A cross-reference table of the program module is provided if the XREF option was specified for the run. The table does not depend upon nor does it automatically generate a module map.

The table contains one entry for each address constant in the module. The left half of the table describes the reference (address constant), showing module location, section name, section offset, and address constant type. The right half of the table describes the external symbol being referenced. Table entries appear in the same sequence as the location of the address constants within the overlay segment.

Figure 28 on page 136 shows a sample cross-reference table. The columns contain the following information:

- CLASS OFFSET - The offset of the address constant relative to the start of the class.
- SECT/PART - The name of the section or part containing the address constant.
- SEG - The segment number if the module is in overlay format.
- RG - The region number if the module is in overlay format.
- ELEMENT OFFSET - The offset of the address constant relative to the start of the section component of the class.
- TYPE - Address constant type. One of six types may appear:
  - V-CON - An adcon normally used for program branching
  - A-CON - An adcon normally used for data reference
  - Q-CON - An adcon that references a pseudoregister or other part by its offset within the class
  - C-LEN - An adcon that will receive the cumulative length of the pseudoregister vector or other class.
  - L TOKE - loader token: represents a unique instance of the module on DASD.
  - R-CON - An adcon referencing the associated data (environment) of the target symbol.
- SYMBOL - The external symbol being referenced.

- SECTION - The name of the section containing the referenced symbol. If the symbol is unresolved or nonrelocatable, this field is set to one of the following:
  - $NON-RELOCATABLE - The address constant contains a nonrelocatable value, such as a pseudoregister offset or PRV length.
  - $UNRESOLVED - The referenced symbol is unresolved.
  - $UNRESOLVED(W) - The referenced symbol is an unresolved weak external reference (WXTRN).
  - $NEVER-CALL - The referenced symbol was identified as never-call.
  - $IMPORTED - The referenced symbol was dynamically resolved.
- SEG - The number of the overlay segment containing the referenced symbol if the module is in overlay format.
- RG - The number of the overlay region containing the referenced symbol if the module is in overlay format.
- ELEMENT OFFSET - The offset of the referenced symbol relative to the start of its containing element, identified by section and class names.
- CLASS NAME - The target class.

The cross reference table contains one segment for each text class containing address constants. A separator line containing the class precedes the adcon listing. Text classes that are not loaded with the module, such as B_PRV, will never contain address constants and will not appear in this report.

```
                              C R O S S - R E F E R E N C E   T A B L E
                              _________________________________________

TEXT CLASS = B_TEXT

--------------  R E F E R E N C E  ------------------------  T A R G E T  --------------------
  CLASS                       ELEMENT                                                ELEMENT
  OFFSET SECT/PART (ABBREV)    OFFSET  TYPE  SYMBOL(ABBREV) SECTION (ABBREV)    OFFSET CLASS NAME

      48 SD1                     48   A-CON                 SD1                    0    B-TEXT
      C4 SD1                     C4   V-CON  LD1            SDX                   A8    B_TEXT
     126 SD1                    126   Q-CON  GFLGA          $NON-RELOCATABLE       0    B_PRV
     18E SD1                    18E   Q-CON  B-TOKEN        $NON-RELOCATABLE      10    B_PRV
     1F6 SD1                    1F6   Q-CON  GFLGC          $NON-RELOCATABLE       2    B_PRV
     25E SD1                    25E   Q-CON  MASTER         $NON-RELOCATABLE       8    B_PRV
     2C6 SD1                    2C6   Q-CON  GFLGE          $NON-RELOCATABLE       1    B_PRV
     32E SD1                    32E   Q-CON  COUNTF         $NON-RELOCATABLE       4    B_PRV
     396 SD1                    396   Q-CON  GFLGG          $NON-RELOCATABLE       3    B_PRV
     3FC SD1                    3FC   CXD                   $NON-RELOCATABLE            B_PRV
     490 SD1                    490   V-CON  SD2            SD2                    0    B_TEXT
     568 SD2                     48   A-CON                 SD2                    0    B_TEXT
     5E4 SD2                     C4   V-CON  LD2            SDX                   AC    B_TEXT
     644 SD2                    124   V-CON  LD3            $PRIVATE               0    B_TEXT
     6A4 SD2                    184   V-CON  LD4            $PRIVATE               4    B_TEXT
     704 SD2                    1E4   V-CON  CM1            CM1                    0    B_TEXT
     7B4 SD2                    294   V-CON  CM1            CM1                    0    B_TEXT
     860 SDX                     48   A-CON                 SDX                    0    B_TEXT
```

Figure 28. Sample binder cross-reference table

Imported and exported symbol table

The Imported and Exported Symbol Table is part of the Module Summary Report. This table is printed if binder options XREF and DYNAM(DLL) are specified and there are symbols to import or export.

The table shows the imported and exported symbols, whether they represent code or data, and, for imported symbols, the name of the dynamic link library from which the symbol was imported.

A sample table is shown in Figure 29. All imported symbols are listed first, followed by the exported symbols. Within each group, symbols are arranged alphabetically. There are some differences between the two groups:

- The member name or z/OS UNIX file name for the IMPORT is derived from the IMPORT control statement.
- The member name for EXPORT is always the same as the symbol name, and so it is omitted.
- Symbol and member names longer than 16 bytes are abbreviated to unique 16-byte replacements that are used in this and other tables. Figure 32 on page 141 shows the correspondence between the long names and their abbreviations.

```
 ***  I M P O R T E D   A N D   E X P O R T E D   S Y M B O L S  ***

IMPORT/EXPORT     TYPE    NAME                MEMBER
-------------     ----    ----------------    ----------------
   IMPORT         CODE    DestroyWindow()     REMSESII
   IMPORT         CODE    ExpandWindow()      REMSESII
   IMPORT         CODE    FinishAcct          REMSESII
   IMPORT         CODE    GenerateColors      REMSESII
   EXPORT         CODE    ShrinkWin-ported
   EXPORT         DATA    ShrinkWin#000001
   EXPORT         DATA    S1
   EXPORT         CODE    S7
   EXPORT         CODE64  __dt__9st-reamFv
   EXPORT         DATA64  __instanc-5Locks
                        *** END OF IMPORT/EXPORT ***
```

Figure 29. Sample binder imported and exported symbols table

Operation summary

The operation summary is generated at the conclusion of the each save or load operation. The save operation summary is produced by entry point IEWBLINK; the load operation summary by entry IEWBLDGO.

The save and load operation summaries are produced when LIST=ALL or LIST=SUMMARY is specified and when meaningful information is available. For example, if the load operation failed, no load summary is produced.

Figure 30 on page 139 and Figure 31 on page 140 contain sample save and load operation summaries. The summaries contain information such as,

- Current processing options These are the binder options in force at the time the module is bound.
- SAVE or LOAD information (as appropriate):
  - Date and time of SAVE
  - Name of output program library
  - Volume serial or storage class of the output program library
  - Name of member
  - Program module attributes (specified and defaulted)

    Note that certain module attributes are not specified as binder options but are determined from the module itself:

- Exceeds 16 MB
- Executable
- Migratable

These attributes provide additional information in the directory entry for later use by the binder or loader.

  - Status (executable/nonexecutable)
  - Total virtual storage required to load the module
  - Load point address of a loaded program module
  - Entry point address of a loaded program module
  - Name of a loaded program module if it has been identified to the system in virtual storage.
- Entry point and alias summary:
  - Main entry point name
  - Alternate entry point and true alias names
  - Addressing modes for main and alternate entry points
  - Classname
  - Class offset
  - Requested alias names that were not assigned
  - Status for alternate entry points and aliases. The status value can be one of the following:

    ADDED
    The name did not exist in the directory and has been added.

    REASSIGNED
    The alias existed in the program module and has been reused in the replacement.

    REMOVED
    The alias existed in the replaced program module, but has not been respecified in the replacement.

    REJECTED
    The name was too long to be saved in the directory or already existed and could not be replaced according to the binder replacement rules.

    STOLEN
    The name existed as an alias to another module, but was reassigned to the module being saved.

    HIDDEN
    The name was added as a result of the ALIASES(ALL) option. AMODE is not listed for hidden aliases.

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(TESTHIGH) STEP(BIND2   ) PGM= IEWBLINK
PROCESSING OPTIONS:

   ALIASES             NO
   ALIGN2              NO
   AMODE               UNSPECIFIED
   CALL                NO
   CASE                UPPER
   COMPAT              UNSPECIFIED
   DCBS                NO
   DYNAM               NO
   EXITS:              NONE
   EXTATTR             UNSPECIFIED
   FILL                NONE
   GID                 UNSPECIFIED
   HOBSET              NO
   LET                 08
   LINECT              060
   LIST                ALL
   LISTPRIV            NO
   MAP                 YES
   MAXBLK              032760
   MSGLEVEL            00
   OVLY                YES
   PRINT               YES
   RES                 NO
   REUSABILITY         UNSPECIFIED
   RMODE               UNSPECIFIED
   STORENX             NO
   TERM                NO
   UID                 UNSPECIFIED
   UPCASE:             NO
   WKSPACE             96K,1024K
   XCAL                YES
   XREF                NO

SAVE OPERATION SUMMARY:

   MEMBER NAME         TSTMOD
   LOAD LIBRARY        PMSBC321.LOADOVLY
   PROGRAM TYPE        PROGRAM OBJECT(FORMAT 1)
   VOLUME SERIAL       1P0303
   DISPOSITION         REPLACED
   TIME OF SAVE        09.37.00  JUL  1, 1997
```

Figure 30. Sample binder save operation summary (Part 1 of 2)

```
SAVE MODULE ATTRIBUTES

   AC                  000
   AMODE               31
   DC                  NO
   EDITABLE            YES
   EXCEEDS 16 MB       NO
   EXECUTABLE          YES
   MIGRATABLE          YES
   OL                  NO
   OVLY                NO
   PACK,PRIME          NO,NO
   PAGE ALIGN          NO
   REFR                NO
   RENT                YES
   REUS                YES
   RMODE               ANY
   SCTR                NO
   SSI
   SYM GENERATED       NO
   TEST                NO
   XPLINK              NO
   MODULE SIZE (HEX)   0009EA78

 ENTRY POINT AND ALIAS SUMMARY:
 NAME:                          ENTRY TYPE    AMODE   OFFSET      STATUS
 IEWBDCTL                        MAIN_EP        31   00011320
 IEWBIND                         TRUE ALIAS     31   00011320   REASSIGNED
 IEWBSTAK                        ALTERNATE     ANY   000676F8   REASSIGNED
```

Figure 30. Sample binder save operation summary (Part 2 of 2)

```
z/OS V1 R5 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(B422735W) STEP(BIND2   ) PGM= IEWBLINK


LOAD OPERATION SUMMARY:

   LOADED NAME         TEST
   TIME OF LOAD        14.00.46  JUNE 30, 1997
   LOAD PT  VADDR(HEX) 00031000
   ENTRY PT VADDR(HEX) 00031000

LOAD MODULE ATTRIBUTES:

  AMODE                24
  PAGE ALIGN          NO
  RMODE                24
  MODULE SIZE (HEX)   00001400
```

Figure 31. Sample binder load operation summary

The Long-symbol abbreviation table

The long-symbol abbreviation table shows the relationships between long symbols and their abbreviations. A long symbol is longer than 16 bytes, and its abbreviation is 16 bytes. The abbreviated symbols are used in several binder reports for better readability.

```
            *** L O N G  S Y M B O L  A B B R E V I A T I O N  T A B L E ***

ABBREVIATION        LONG SYMBOL

__ct__9Ex-lassFv := __ct__9ExpoClassFv
__dt__9Ex-lassFv := __dt__9ExpoClassFv
__sinit80-____Fv := __sinit80000000__dfpft_worklib_source_c_x955404e___Fv
__sterm80-____Fv := __sterm80000000__dfpft_worklib_source_c_x955404e___Fv
an_object-456789 := an_object01234567890123456789012345678901234567890123456789


*** E N D  O F  L O N G - S Y M B O L  A B B R E V I A T I O N  T A B L E ***
```

Figure 32. Sample binder long-symbol abbreviation table

Short mangled name report

The list of abbreviated names was expanded to display mangled names. This list was designed to avoid the repetition of data and to keep the Mangled name abbreviation and the DeMangled name together. This list is already sorted in abbreviated name order.

A new list was added to account for the demangling of short names which normally do not require an abbreviation, for example, names less than seventeen bytes. Although there are probably few of these names, an accounting must be made for them.

The changes were made to cause no listing changes if there are no Mangled names to be displayed.

```
   ********     S H O R T   M A N G L E D   N A M E S     ******

MANGLED NAME           DEMANGLED NAME

__javPshort       ==   __javPshort


   ********   E N D   S H O R T   M A N G L E D   N A M E S   ****
```

Figure 33. Sample binder short mangled name report

Abbreviation/Demangled name report

The abbreviation report has been expanded to provide a cross reference to the DeMangled names.

```
   **  A B B R E V I A T I O N  /  D E M A N G L E D   N A M E S **

ABBR/MANGLE NAME    LONG SYMBOL

__addr_34-tring) := __addr_34_java/lang/IllegalArgumentExceptionI6_<
                    init>(L16_java/lang/ String)
__javCls1-ension := __javCls18_java/awt/Dimension
  $$DEMANGLED$$  == java.awt.Dimension
__javCls1-nuItem := __javCls17_java/awt/MenuItem
  $$DEMANGLED$$  == java.awt.MenuItem
__jav15_j-ame()V := __jav15_java/awt/Button9_buildName()V
  $$DEMANGLED$$  == void java.awt.Button.buildName()
__jav15_j-ener)V := __jav15_java/awt/ButtonY17_addActionListener(L29
                    _java/awt/event/ActionListener)V
  $$DEMANGLED$$  == synchronized void java.awt.Button.addActionListe
                  + ner(java.awt.event.ActionListener)
__jav15_j-hics)V := __jav15_java/awt/Canvas5_paint(L17_java/awt/Grap
                    phics)V
  $$DEMANGLED$$  == void java.awt.Canvas.paint(java.awt.Graphics)

   ***   E N D   A B B R E V / D E M A N G L E D   N A M E S   ***
```

Figure 34. Sample binder abbreviation/demangled names report

Notes:

1. Demangled Names always are preceded with $$DEMANGLED$$.
2. "==" is always followed by the demangled name.
3. Continuation lines for demangled names are prefixed by "+".
4. The demangled name always follows the related abbreviated and mangled names.
5. Reports are in alphabetical order by mangled Name / abbreviation.
6. Names which cannot be demangled are omitted from the list. No message is provided.
7. There are two messages which may appear under the demangled name heading, within the list:
   - Unable to CONTINUE DEMANGLE = Abnormal Termination in the Demangler. No further demangling is attempted. It also causes message IEW2441I MANGLED NAMES EXIST- UNABLE TO ACCESS DEMANGLER to be written following the report,
   - Demangled Name greater than 16384 bytes = Very long demangled name was encountered. The name is not printed.

DDname versus Pathname cross reference report

The pathname to DDname table will be printed even if the binder map is not printed. Since the constructed DDnames (such as '/0000003') are used in error messages, if a map is not requested or if not map is produced because the save or load does not complete, you have no way of determining which z/OS Unix files has been referenced. This report allows you to make that correlation. The following is an example of a DDname vs. pathname report.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|  D D N A M E   V S   P A T H N A M E   C R O S S   R E F E R E N C E  |
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
DDNAME    PATHNAME
------    ------------------------------------------------------------
0000001   /PM64B301/dlla07
           *** END OF DDNAME VS PATHNAME
```

The message summary report

The message summary report provides a table of unique message numbers issued by the binder. Messages are separated by severity. Message numbers are counted even if the message was suppressed by the message exit or the MSGLEVEL option.

You can use message numbers from this report to scan the Input Event Log for messages of interest. This is particularly helpful when modules are batched and listings are extensive.

When the Binder is required to print a message containing a variable (symbol) whose length is greater than 1024 bytes, the message will print only the first 1020 bytes of the variable(symbol). When this occurs, the message will contain an asterisk in the blank column immediately following the message number. Additionally, a note will be printed immediately following the message summary report indicating at least one message has had a variable (symbol) truncated.

```
z/OS V1 R3 BINDER     hh:mm:ss ddddd mmmmm dd, yyyy
BATCH EMULATOR  JOB(PM64B251) STEP(BIND1   ) PGM= IEWBLINK
IEW2322I 1220  811      INCLUDE         DD1(PO2)
IEW2308I*1112 SECTION
 4KLG_SD14KLG_SD1LONG_SD1LONG_SD1LONG_SD1LONG_50X010123456789012345678901
 012345678901234567890123456789012345678901234567890123_50X030123456789012345678901234567890
 01234567890123456789012345678901234567890123_50X050123456789012345678901234567890123456789
 01234567890123_50X070123456789012345678901234567890123456789012345678901234567890123_50X080
 0123_50X090123456789012345678901234567890123456789012345678901234567890123_50X1001234567890
 012345678901234567890123456789012345678901234567890123_50X120123456789012345678901234567890
 012345678901234567890123456789012345678901234567890123_50X140123456789012345678901234567890
 012345678901234567890123_50X160123456789012345678901234567890123456789012345678901234567890
 01234567890123_50X180123456789012345678901234567890123456789012345678901234567890123_50X190
 0123_50X200123456789012345678901234567890123 HAS BEEN MERGED.
 -  -  -  -  -  -  -  -  -  -  1345 LINE(S) NOT DISPLAYED

----------------------
MESSAGE SUMMARY REPORT
----------------------
 SEVERE MESSAGES        (SEVERITY = 12)
 NONE

 ERROR MESSAGES         (SEVERITY = 08)
 2333

 WARNING MESSAGES       (SEVERITY = 04)
 NONE

 INFORMATIONAL MESSAGES (SEVERITY = 00)
 2008  2013  2278  2308  2322

 *** NOTE: ANY MESSAGE WITH AN '*' FOLLOWING
     THE MESSAGE NUMBER MEANS A VARIABLE IN THAT
     MESSAGE WAS TRUNCATED TO 1020 BYTES.

 **** END OF MESSAGE SUMMARY REPORT ****
```

Figure 35. Message summary report (variable truncated)

Chapter 9. Binder serviceability aids

There are several diagnostic aids that can be used to analyze and resolve problems found while using the Program Management binder. These include:

- Binder output data sets
- The AMBLIST service aid
- The IDCAMS printing utility

This chapter also explains how to diagnose information when invoking the binder from the z/OS UNIX shell using the c89 command.

The complexity of the problem being analyzed dictates the number and combinations of the above aids needed in order to solve the problem. The following discusses each of the aids listed above.

Binder output data sets

The program management binder generates various output listings, which supply you with diagnostic information at different levels of specificity. The data sets containing this information can be specified in the JCL, at the time the binder is invoked in batch mode, or in the STARTDialog API call, when the binder is invoked interactively.

Table 7 shows the output data sets by DDNAME, and briefly explains the purpose of their contents. A more specific description of each data set follows the table.

Binder output data sets and their contents

Table 7. Binder data sets and their contents

DD name	Contents
SYSPRINT	Depending on user-specified options, this data set may contain binder processing messages, a data map of the program object or load module, a cross-reference list depicting numerical offsets of the elements within a class of binder data, and other information.
IEWDIAG	In the absence of SYSPRINT's allocation, this data set receives all the messages that would have gone to SYSPRINT. This may be the case if the binder is invoked interactively via its API.
IEWTRACE	If specified, this data set contains tracing information as control is passed from one binder module to another. Input and/or output data, as well as return codes, are echoed in most tracing entries, making it easier to follow and diagnose binder processing events.
IEWDUMP	The information in this data set represents a snapshot of binder data in its internal organization. When the information in the above data sets is not sufficient to troubleshoot a problem, this information becomes necessary. Data is directed to this data set when there is an abnormal termination in the binder's processing, or when a caller makes a request for a dump upon entry to a specific binder module.

Table 7. Binder data sets and their contents (continued)

DD name	Contents
IEWGOFF	This data set contains the Generalized Object File Format (GOFF) records produced by the binder when its input is Extended Object (XOBJ) module records, which are generated by some compilers. Once built in storage, the GOFF records are processed and bound by the binder. If this data set is specified at the time the binder is invoked, the produced GOFF records will be echoed to it. Should the binder encounter any problems processing the GOFF records, this data set may be useful in diagnosing problems in the XOBJ-to-GOFF conversion process or in the source XOBJ records.

The SYSPRINT data set

Interpreting the contents of SYSPRINT

The specification of this data set is required during the batch invocation of the binder. It is optional in the binder's API mode. The output contained in this data set is organized into several informational categories, the number of which depends on the options specified during the binder invocation. These categories are:

- Header
- Input Event Log
- Program Module Map
- Renamed Symbol Table
- Cross-Reference Table
- Imported and Exported Symbol Table
- Long-name Cross-Reference Table
- Operation Summary
- DDNAME versus Pathname Cross-Reference Report
- Message Summary

A brief description of each of these categories is given below. See Chapter 8, "Interpreting binder listings," on page 127 for complete descriptions and samples of all the categories.

Header: The header is written at the beginning each section of the output. The header contains information on the release and modification level and on how the binder was invoked.

- Name, version, release, and modification level of the binder
- Time, day, and date of invocation
- Job name, step name, program name, and (if one has been used) procedure name when invoked by use of a batch interface.
- Binder entry point name.

Input event log: The input event log is a chronological log of the events that took place during the input phase of binder operation. Its presence is controlled by the LIST option. If LIST(OFF) or NOLIST is specified, no input event log is generated. If LIST(STMT), LIST, or LIST(SUMMARY) is specified, only input events pertaining to control statements are logged. If LIST(ALL) is specified, all input events are logged (such as those initiated by binder function calls as well as those initiated by control statements).

Program module map: A map of the program module is generated if the MAP option was specified at the binder invocation. The module map shows a layout of the binder data as well as source DDNAMEs from which data was extracted in order to resolve references and bind a module. This map is often used in conjunction with the output of the service aid AMBLIST in order to compare data layouts from this map and AMBLIST's so that anomalies can be detected.

Renamed symbol table: The binder normally processes symbols exactly as received by the compiler. However, certain symbolic references generated by C or C++ compilers may be renamed by the binder, if they contain long or mixed case names (L-names) and cannot be resolved using the L-name during autocall. During renaming, the L-name reference is replaced by its equivalent short-name abbreviation. Such replacements, whether resolved or not, will appear in the Renamed Symbol Table.

Cross-reference table: The cross-reference is provided if the XREF option was specified at the invocation of the binder. The table does not depend upon nor does it automatically generate a module map.

The table contains one entry for each address constant (ADCON) in the module. The entry shows such information as the type of ADCON (V-CON,A-CON,Q-CON,CXD), its offset within a class and a section, etc.

Imported and exported symbol table: This table is produced when the binder option DYNAM(DLL) is specified and a program object produced by the binder is to import or export symbols during dynamic binding.

Long-name cross-reference table: When the binder processes symbol names that are longer than 16 characters, it generates unique abbreviations for these long names. Such abbreviations are used in some output reports, such as the "Program Module Map" and the "Cross-Reference Table", in order to make the reports more readable. The "Long-name Cross-Reference Table" simply shows the relationship between the long names and their abbreviations.

Operation summary: The operation summary is generated at the conclusion of each save or load operation. The save operation summary is produced if you invoked the binder at entry point IEWBLINK. The load operation summary is produced if you invoked the binder at entry point IEWBLDGO.

The save and load operation summaries are produced when LIST=ALL or LIST=SUMMARY is specified and when meaningful information is available. For instance, if the load operation failed, no load summary is produced.

DDNAME versus pathname cross-reference report: This report is printed even if the MAP is not printed. Since the constructed DDNAMEs (such as '/0000003') are used in error messages, there would be no way of knowing the z/OS UNIX file name without this report

Message summary: The Message Summary provides a table of unique message numbers issued by the binder. Messages are categorized by severity. Message numbers are counted even if their corresponding message text was suppressed by the message exit or the MSGLEVEL option.

You can use message numbers from this report to scan the Input Event Log for messages of interest. This is particularly useful when bindings are batched and output listings are extensive.

Allocating the SYSPRINT data set

This data set can be either a SYSOUT data set, a sequential data set, or a member of a partitioned data set. The data set attributes should be:

```
DSORG=PS,RECFM=FBA,LRECL=121
```

BLKSIZE can be equal to or larger than the LRECL. IBM recommends omitting BLKSIZE so as to take advantage of an optimal, system-determined block size.

The IEWDIAG data set

Interpreting the contents of IEWDIAG

This data set is useful for obtaining diagnostic information if there is no SYSPRINT data set. In this case, the phrase ″diagnostic information″ merely refers to the messages which would normally be written to SYSPRINT. This would commonly be the case if the binder is being invoked from a utility via the API.

Allocating the IEWDIAG data set

This data set can be either a SYSOUT data set, a sequential data set, a member of a partitioned data set, a USS file, or a TSO terminal. The data set attributes are the same as those for SYSPRINT.

The IEWTRACE data set

TRACE option

The binder TRACE option may be specified as:

```
TRACE=ALL|OFF|(start_ecode,[end_ecode]
```

By default, the option is set to TRACE=ALL. With this setting, all trace entries will be written if the IEWTRACE DD is allocated. TRACE=OFF will suppress all tracing.

The TRACE data set may become extremely large. It may be useful to specify that only some of the trace entries be written out, by using selective trace. To do this, code the trace option as:

```
TRACE(start_ecode,[end_ecode])
```

TRACE will turned on when 'start_ecode' is seen (as if TRACE=ALL had been specified at that point). If 'end_ecode' is specified, TRACE will be turned off when 'end_ecode' is seen (as if TRACE=OFF had been specified at that point).

Interpreting the contents of IEWTRACE

The contents of this data set represent cumulative tracing entries issued by the binder's modules during their processing sequence. Trace entries are produced at entry to and exit from each module, as well as at other points deemed important for diagnosis purposes. For instance, most binder modules produce trace entries whenever they request a system service. This information proves useful to IBM when servicing the binder.

All the entries in a trace data set are numbered, as can be seen in the sample trace in Figure 36 on page 149. Each entry begins with a sequence number and consists of one or more lines. The four alpha characters following the sequence numbers represent the last four letters in a binder's module name, all of which begin with ″IEWB″. For instance, the module name in trace entry 0 is ″IEWBOGET″. Horizontally, the next eight numeric (hexadecimal) digits represent internal codes which signify the events taking place in a module (the coined term to refer to these codes is ″event codes″, or ″ecodes″, for short). So, for example, the ecode in trace

entry 0 means ″entry to module IEWBOGET″, and the ecode in entry 1 means ″exit from IEWBOGET″. In entry 1, the ecode at the far right means that the ″processing in IEWBOGET was successful.″ A complete list of ecodes and their definitions is available to the organization servicing the binder, but a general guideline for interpreting such ecodes is given below, under “Interpreting binder ecodes.”

One or more lines in a trace entry provides all the pertinent diagnostic information at the time the trace was issued. For instance, most module exit entries print the return and reason codes returned to the calling module. In entry 23, module IEWBXIOP exited (ecode D2A1A100) with a return code of 12, in deference to entry 24, where IEWBXR00 exited (ecode 409FA100) with a return code of 4.

Finally, the characters between the two parenthesis in each entry is an internal time-stamp.

```
00000000   OGET   B903A000 (13:33:48.223045)
                0013 X
00000001   OGET   B904A100 (13:33:48.223046) B900B000
00000002   SGET   C400A000 (13:33:48.223049)
            0000000316 D
00000003   SGET   C402A100 (13:33:48.223050)
            0000000316 D
            00000000 X
            000188D0 X
00000004   RCRE   EA20A200 (13:33:48.223053)
            ABCDEFGHIJKLMNOPQRSTUVWXYZ
            T
            00000000 X
            00000000 X
00000005   RSDM   ED00A000 (13:33:48.223056)
            BRIO_PTR =
            000188D0 X
00000006   RSDM   ED21A200 (13:33:48.223056)
00000007   XR00   4090A000 (13:33:48.223058)
00000008   XIOP   D2A0A000 (13:33:48.223061)
            SYSPRINT
00000009   XIOP   D2A1A100 (13:33:48.223062) D000B000
00000010   XR00   409FA100 (13:33:48.223063) 4000B000
00000011   RSDM   ED22A601 (13:33:48.225296)
00000012   RSDM   ED26A602 (13:33:48.225297)
00000013   RSDM   ED23A200 (13:33:48.225297)
00000014   RSDM   ED01A100 (13:33:48.225298)
00000015   RCRE   EA21A200 (13:33:48.225298)
00000016   SGET   C400A000 (13:33:48.225302)
            0000000524 D
00000017   SGET   C402A100 (13:33:48.225304)
            0000000524 D
            00000000 X
            00018A10 X
00000018   CLCK   F200A001 (13:33:48.225309)
```

Figure 36. Trace Sample

Interpreting binder ecodes: Although supplying a complete list of binder ecodes is beyond the scope of this document, providing a general guideline for reading such ecodes is necessary and may prove useful when trying to diagnose a binder problem.

An ecode is a fullword bit string in the hexadecimal format MMEEGGGG. The three subfields are used as follows:

- MM - Module identifier (00-FF). It identifies the module in which the event took place.
- EE - Event number within the module (00-FF).
- GGGG - Generic event code. This number varies as follows:

```
GGGG                        meaning
-------------------------------------------
A0XX                        Module entry.  XX is usually 00, but
                            if a module has multiple entry points,
                            it may be 01, 02, etc.

A1XX                        Module exit.   XX is usually 00, but
                            if a module has multiple exit points,
                            it may be 01, 02, etc.

B000                        Returned to caller, trace, etc

XXXX                        Message number of associated message
```

All modules have both an entry and an exit trace record, and the exit trace record gives the return and reason codes. Most modules also trace calls for entry and return to system services.

The following specific ecodes may be of help:

- FFA6B000

  Contains a copy of a message to be issued (some of these messages might not actually appear in SYSPRINT because of the MSGLEVEL setting).
- 0040XXXX-005CXXXX

  Trace parameters passed on binder API calls.
- A200A001/A200A101

  Trace additions of symbol names to the binder's Namelist. Contains the name, its category code, and the assigned name list index.
- 8000A000

  Traces the addition of an element index record to the binder's workmod. It contains the pertaining class and section names.

There is normally a DEND entry at the end of the trace of a complete binder execution. If it is not there, the trace was truncated due perhaps to a program check in the binder. In this case, the trace would probably not be very useful as it would not show the complete binder logic sequence.

If you know that the binder did not end normally, then backing up from the DEND entry may show a binder terminal error message. For normal termination you will see the IEW2008I message.

Allocating the IEWTRACE data set

This information is generated whenever the IEWTRACE ddname is specified in the batch mode of the binder, or when the TRACE file name is specified in the FILES parameter of the STARTDialog API call. In batch mode, this data set can be either a SYSOUT data set, a sequential data set, or a member of a partitioned data set. This data set cannot be a USS file because it has variable length records with binary fields. The DCB attributes for this data set should be:

```
DSORG=PS,RECFM=VB,LRECL=84
```

Note that RECFM can be VBA as well. BLKSIZE can be any multiple of 4 which is equal to or larger than the LRECL, 84. IBM recommends omitting BLKSIZE so as to take advantage of an optimal, system-determined block size.

The IEWDUMP data set

The information in this data set represents a snapshot of binder data in its internal organization. When the information in the other diagnosis data sets is not enough to identify a problem, this information becomes essential. For problems that occur within the binder, IEWDUMP or SYSUDUMP is sufficient and easier to work with than an IPCS format dump.

Generating a dump in the binder

Data is directed to this data set when there is a terminal (abnormal) error in the binder, when a caller makes a request for a dump upon entry to a specific binder module, or when a program check or system abnormal termination occurs while in the binder.

If SYSUDUMP or SYSABEND has been allocated, a SYSUDUMP will be taken if a binder logic error or a program check or system abend occurs. If IEWDUMP has been allocated, a dump which contains formatted binder control blocks and the dataspace storage in use by the binder will be produced. (You would get both dumps if SYSUDUMP and IEWDUMP were both allocated). Logic errors are terminal and the binder job will terminate after taking the dump.

You can request that a formatted dump (IEWDUMP) be taken when a specific non-terminating binder event code (ecode) is seen. In this case, binder execution will continue after the dump. To request that a dump be taken on a specific ECODE in batch mode, the following is a JCL example:

```
//LINK       EXEC PGM=IEWBLINK,PARM=('LET(8)',XREF,
//           'DUMP=''45082508''',MAP)
```

To request a dump on a specific ecode using the binder interface, use the following assembler example as a guide.

```
*******************************************
*        START THE BINDER DIALOG            *
*******************************************
STARTD   IEWBIND FUNC=STARTD,RETCODE=RETCODE,RSNCODE=RSNCODE,               X
               DIALOG=DTOKEN,OPTIONS=OPTLIST,FILES=FILELIST
*
OPTLIST  DS    0F
         DC    F'2'                          NUMBER OF ENTRIES IN OPTIONS LIST
         DC    CL8'MSGLEVEL',F'2',A(MSGVALU)
         DC    CL8'DUMP    ',F'10',A(ECODE)  DUMP ON SPECIFIC ECODE
MSGVALU  DC    C'12'
ECODE    DC    C'''2500A000'''               ECODE FOR ENTRY TO
*                                            BINDER MODULE IEWBFMOD
FILELIST DS    0F
         DC    F'1'                          NUMBER OF ENTRIES IN FILES LIST
         DC    CL8'DUMP    ',F'8',A(DDNAME)  DUMP DATA SET REQUESTED
DDNAME   DC    C'IEWDUMP '
```

Interpreting the contents of IEWDUMP

The formatted portion will be at the end of the dump. For each workmod, the workmod index records are shown, followed by Namelist entries.

Workmod data elements: Module data in the binder internal (workmod) format is organized into units called elements. (Some older or obsolete binder documentation may call these 'items' or even 'itemids'). An element is identified by a section name and class name.

The formatted portion of the dump provides the information necessary to find the data associated with each element in each workmod (see Figure 37 for an example). The data is formatted in a three-level hierarchy as follows:
- workmod
- section
- class

The first line output for each element prints:

APPPTR
The pointer to the first ″append pointer″ - that is, to the control block describing the first block of contiguous data in the element.

CNT The append count (the total number of such contiguous blocks).

HIW "HI-WATER" - the highest record number in the element. For text, this is the last byte of initialized text - it may be smaller than the total CSECT text length.

LRECL
Length of one logical record

In the second line for each element, 20 bytes of attribute information are shown. The first two fields give the offset of the data within the containing class and the length, relative to records. (For text, the length of one record is one byte.)

```
z/OS           PROGRAM MANAGEMENT DIAGNOSTICS

WORKMOD TOKEN:        0 21EDBFB0

 SECTION:    printf
   CLASS:    B_ESD
       APPPTR: 21F23620 CNT:        1 HIW:        3 LRECL:       48
       CLASS ATTRIBUTES:   0000008A 00000003 00480000 40100000 00000000
   CLASS:    B_IDRL
       APPPTR: 21F25720 CNT:        1 HIW:        1 LRECL:       10
       CLASS ATTRIBUTES:   00000007 00000001 00100000 40100000 00000000
   CLASS:    B_TEXT
       APPPTR: 21F21D78 CNT:        1 HIW:        A LRECL:        1
       CLASS ATTRIBUTES:   000001E0 0000000A 00010303 00000001 00000000
```

Figure 37. IEWDUMP sample – Workmod token area

Finding the data in the dump: Go to address APPPTRT to find the data in an element. The important fields are:

Table 8. APPPTRT dump data

Offset (HEX)	Content
0	Next append control block
4	Starting offset of the data described by this block from the start of the containing element
8	Count of logical records described by this block
C	Data pointer - location of actual data

Table 8. APPPTRT dump data (continued)

Offset (HEX)	Content
C	Type of pointer (1 = virtual addr, 2 = dataspace)
10	Alet
14	Virtual address

Allocating the IEWDUMP data set

This information is generated whenever the IEWDUMP DDNAME name is specified in the batch mode of the binder, or when the DUMP file name is specified in the FILES parameter of the STARTDialog API call. This data set can be either a SYSOUT data set, a sequential data set, a member of a partitioned data set, a USS file, or a TSO terminal. If it is a USS file, also code DATATYPE=TEXT.

```
DSORG=PS,RECFM=VB,LRECL=125
```

Note that the BLKSIZE can be equal to or larger than the LRECL, 125. IBM recommends omitting BLKSIZE so as to take advantage of an optimal, system-determined block size.

The IEWGOFF data set

Interpreting the contents of IEWGOFF

This data set contains the Generalized Object File Format (GOFF) records produced by the binder when its input is Extended Object (XOBJ) module records, which are produced by specifying the RENT option in the C/C++, OO Cobol, and other compilers. Once built in storage, the GOFF records are processed and bound by the binder. The records in this data set are merely a snapshot of the records produced during a binder run. If the binder encounters any problem processing them, it may be useful to look at the GOFF records in this data set so as to diagnose problems in the XOBJ-to-GOFF conversion process or in the source XOBJ records. For this reason, the contents of this data set may be requested by the IBM organization servicing the binder.

See GOFF records and their formats in *z/OS MVS Program Management: Advanced Facilities*.

Allocating the IEWGOFF data set

If XOBJ records are passed to the binder as input and the IEWGOFF ddname is specified in the JCL, GOFF records are written to the indicated data set. The IEWGOFF data set can be either a sysout data set, a sequential data set, or a member of a partitioned data set. It cannot be a USS file. The attributes of the GOFF data set should be:

```
DSORG=PS,RECFM=VB,LRECL=2124
```

Note that the BLKSIZE can be a multiple of 4 equal to or larger than the LRECL, 2124. IBM recommends omitting BLKSIZE so as to take advantage of an optimal, system-determined block size.

The AMBLIST service aid

AMBLIST is useful and even essential in many cases. However, there are a few limitations that you should be aware of.

1. AMBLIST does not display all the internal control blocks of program objects. Therefore, AMBLIST's output may not be sufficient to diagnose a problem which requires knowledge of such information.
2. If there is anything wrong with the module (program object or load module), AMBLIST may fail. Sometimes specifying OUTPUT=MODLIST in the AMBLIST job will help in this situation, since the XREF portion of the output is highly dependent on all the cross links between ESD and RLD being correct.

Here are three JCL examples for the invocation of AMBLIST:

```
//EXAMPLE1   EXEC   PGM=AMBLIST,REGION=16M
//SYSPRINT   DD     SYSOUT=*
//LOADLIB1   DD     DSN=APPS.PDSE,DISP=(SHR)
//SYSIN      DD     *
    LISTLOAD  DDN=LOADLIB1,MEMBER=(APP1)
/*
//EXAMPLE2   EXEC   PGM=AMBLIST,REGION=16M
//SYSPRINT   DD     SYSOUT=*
//LOADLIB2   DD     DSN=GAMES.PDSE,DISP=(SHR)
//SYSIN      DD     *
    LISTLOAD  DDN=LOADLIB2,MEMBER=(APP1),OUTPUT=MODLIST
/*
//EXAMPLE3   EXEC   PGM=AMBLIST,REGION=16M
//SYSPRINT   DD     SYSOUT=*
//HFS1       DD     PATH='/u/userid/main',PATHDISP=(KEEP,KEEP)
//SYSIN      DD     *
    LISTLOAD  DDN=HFS1,OUTPUT=MODLIST
/*
```

For more information, see AMBLIST, in *z/OS MVS Diagnosis: Tools and Service Aids*.

The IDCAMS printing utility

You can use IDCAMS to print the contents of a program object in a USS file, or the unformatted contents of a program object in an MVS™ data set.

For USS files you must use OCOPY to copy the file to a sequential native MVS data set before using IDCAMS.

An example of the IDCAMS JCL follows:

```
//DUMPMOD  EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//INPUT2 DD DSN=PDSE1.APPS(APP1),DISP=SHR
//SYSIN DD *
 PRINT INFILE(INPUT2)
/*
```

Diagnostics when the binder is invoked from the c89 command

Use the following steps to obtain diagnostic information when the binder is invoked from c89:

1. Make sure the IEWDUMP and IEWTRACE data sets are pre-allocated and cataloged. The lowest-level qualifier must be the same as the DDNAMEs (IEWTRACE or IEWDUMP), and the remainder (prefix) must be the same for both.
2. In the UNIX shell, export **_C89_DEBUG_PREFIX=your_prefix**, where **your_prefix** is the prefix used on the names of the IEWDUMP and IEWTRACE data sets. The c89 command will do the allocations (DISP=SHR) during the bind step.

Guideline: For problem diagnosis, it is strongly advised that you also use **export _BPXK_JOBLOG=2**, so that any message appearing on the operator console is also written to stderr. In addition to IEWDUMP and IEWTRACE, use both the c89 -v and -V options to capture stdout and stderr. This method makes the binder invocation parameters and the binder output listing with error messages available for diagnosis.

Appendix A. Using the linkage editor and batch loader

All of the services of the linkage editor and batch loader can be performed by the program management binder. We recommend that you convert to exclusive use of the binder. However, if you do need to use the linkage editor or batch loader, most of the information in this document is applicable with a few differences. This appendix describes those differences.

Creating programs from source modules

AMODE and RMODE differences

The differences in linkage editor processing of AMODE and RMODE values are:

- Values of *MIN* or 64 for AMODE are not supported.
- If only one value, either AMODE or RMODE, is specified on the MODE control statement or on the AMODE and RMODE options, the other value is implied according to the following table:

Value specified	Value implied
AMODE=24	RMODE=24
AMODE=31	RMODE=24
AMODE=ANY	RMODE=24
RMODE=24	see note below
RMODE=ANY	AMODE=31

Note: If only an RMODE of 24 is specified, no overriding AMODE value is assigned. Instead, the AMODE value in the ESD data for the main entry point, a true alias, or an alternate entry point is used in generating its respective directory entry.

- When building an overlay format load module, the AMODE and RMODE values in the ESD data of the output module are discarded and can be restored only by including the object modules carrying those values.
- ESD records that specify AMODE(ANY) RMODE(ANY) are handled differently:
  - If the entry point external symbol is marked AMODE ANY/RMODE ANY, associated entry point attributes are assigned according to the following hierarchy:
    - If the load module contains one or more CSECTs marked AMODE 24, the linkage editor assigns an AMODE of 24 to all entry points that have ESD entries marked AMODE ANY/RMODE/ANY.
    - If the load module has an RMODE of 24 and it contains no CSECTS marked AMODE 24, the linkage editor assigns an AMODE of ANY to these entry points.
    - If the load module RMODE is ANY, the linkage editor assigns an AMODE of 31 to these entry points.

Unsupported input module formats and contents

The linkage editor and batch loader do not support GOFF or XOBJ object module formats, program objects, or object modules with 64-bit content, nor do they support z/OS UNIX files. The batch loader does not accept control statement input.

Invoking the linkage editor and batch loader

You can invoke the linkage editor and batch loader with JCL, under TSO, or through a program.

Invoking the linkage editor and batch loader with JCL

The linkage editor and batch loader programs can be invoked on the PGM parameter of the JCL EXEC statement.

The linkage editor is invoked using the program name HEWLKED. The linkage editor can also be invoked by the following aliases: HEWLF064, IEWLF440, IEWLF880, and IEWLF128. This program link-edits a load module and stores it in a partitioned data set library.

The batch loader is invoked using the program name HEWLDIA. This program link-edits a load module, loads it into virtual storage, and executes it.

SYSLIN data sets

The maximum block size of data sets defined in the SYSLIN definition is 3200 bytes. The linkage editor does not support load modules or program objects in the primary input. The batch loader does not support program objects in the primary input.

SYSPRINT and SYSLOUT data sets

The DCB parameters for SYSPRINT and SYSLOUT need not be specified. If they are specified, they must be RECFM=FA or RECFM=FBA and LRECL=121, and the BLKSIZE parameter is any multiple of 121 to a maximum of 4840 bytes.

See “Invoking the binder with JCL” on page 31 for information on using JCL.

SYSUT1 data set

In addition to the required data sets described in “Binder DD statements” on page 33, the linkage editor uses another data set to hold data records during processing. The linkage editor places intermediate data in this data set when storage allocated for input data or certain forms of out-of-sequence text is exhausted.

A SYSUT1 DD statement is required to describe this data set. It must be a sequential data set assigned to a single direct access storage device. Space must be allocated for this data set, but the data set characteristics are supplied by the linkage editor.

Message IEW0294 will be issued if you specify more than one volume.

Included data sets

If an included data set contains another INCLUDE statement, the specified module is processed but any data following the INCLUDE statement is not processed.

Concatenated data sets

All of the data sets in a concatenated list must have the same record characteristics (format, record length). Concatenated data sets can have differing block sizes and be in any order of blocksize.

All concatenated call libraries must be of the same type (object modules or load modules). A call library cannot contain program objects.

Invoking the linkage editor from a program

Programming Interface information

You can pass control to the linkage editor from a program using the LINK, ATTACH, LOAD, CALL, and XCTL macros using either 24-bit or 31-bit addressing. You must supply a save area address in register 13.

The linkage editor is invoked using the HEWLKED program name, or one of these aliases: HEWLF064, IEWLF440, IEWLF880, or IEWLF128.

The use of these macros is identical to usage for the binder with the exception of the ddname list passed as a parameter on LINK, ATTACH, CALL, and XCTL calls.

The sequence of the 8-byte entries in the ddname list for the linkage editor is as follows:

Entry	Alternate Name For:
1	SYSLIN
2	Member name (The name under which the output load module is stored in the SYSLMOD data set. This entry is used if the name is not specified on the SYSLMOD DD statement or if there is no NAME control statement.)
3	SYSLMOD
4	SYSLIB
5	Not applicable
6	SYSPRINT
7	Not applicable
8	SYSUT1
9-11	Not applicable
12	SYSTERM

When the linkage editor completes processing, a return code is returned in register 15 (see “Linkage editor return codes” on page 172 for a list of linkage editor return codes).

End of Programming Interface information

Invoking the batch loader from a program

Programming Interface information

You can pass control to the batch loader from a program using the LINK, ATTACH, LOAD, CALL, and XCTL macros using either 24-bit or 31-bit addressing. You must supply a save area address in register 13.

The batch loader can be invoked at three different entry points to perform the following services:

HEWLDIA
: Link-edits a load module, loads it into virtual storage, and executes it.

HEWLDI
: Link-edits a load module, loads it into virtual storage, and identifies it. HEWLDI returns the address of an 8-character module name in register 1. This name can be used to invoke the loaded program using a LINK or ATTACH macro.

HEWLD

Link-edits a load module and loads it into virtual storage, but does not identify it. HEWLD returns the entry point of the loaded module in register 0 (the high order bit is on for AMODE). Register 1 points to two fullwords. The first points to the beginning of storage occupied by the loaded program, and the second contains the length of the loaded program.

The ATTACH, LINK, LOAD, and XCTL macros are described in *z/OS MVS Programming: Assembler Services Guide*. The use of these macros is identical to usage for the binder, with the exception of the ddname list passed as a parameter on LINK, ATTACH, CALL, and XCTL calls.

The sequence of the 8-byte entries in the ddname list for the batch loader is as follows:

Entry	Alternate Name For:
1	SYSLIN
2	Not applicable
3	Not applicable
4	SYSLIB
5	Not applicable
6	SYSLOUT
7-11	Not applicable
12	SYSTERM

The batch loader generates a return code when it completes its execution and returns it in register 15. See “Batch loader return codes” on page 174 for more information on batch loader return codes.

End of Programming Interface information

Invoking the linkage editor and batch loader under TSO

You also use the LINK command to invoke the linkage editor and the LOADGO command to invoke the batch loader under TSO. If you specify the NOBINDER option on either of these commands, the linkage editor or batch loader will be invoked rather than the binder.

Editing a control section

Replacing control sections

A restriction applies when you request the linkage editor to perform both a CHANGE and a REPLACE operation on the same included module. This situation occurs when you delete one or more control sections *and* rename references to symbols within a removed control section to some other external symbol all within the scope of a single INCLUDE. When you change more than one entry name within a removed control section to a single new external symbol, you must specifically include the control section that resolves the new external symbol prior to the CHANGE operation.

If a replaced control section contains unresolved external references and the replacing control section does not, you must either specify the NCAL parameter, use the REPLACE statement to delete the unresolved external references, or use the LIBRARY statement to mark the references for restricted no-call or never-call.

Deleting an external symbol

If you use the linkage editor to delete a control section that contains any unresolved external references, those references are NOT removed from the external symbol dictionary.

If the input does not have an INCLUDE statement or object module after a REPLACE statement that is to delete a CSECT, and there are external references to be resolved from SYSLIB, the linkage editor causes the delete request to operate on the first module from SYSLIB and deletes the control section.

Control statement reference

Continuing a statement

You indicate that a control statement line is continued onto the next line by placing a non-blank character in column 72 of the line. The continued statement must begin in column 16 of the next line.

ALIAS statement

No more than 64 alias names can be assigned to one load module.

CHANGE statement

If a CHANGE statement is not followed by any included module, the linkage editor applies the change to the first module, if any, brought in during automatic library call.

ENTRY statement

If you provide more than one ENTRY statement, the main entry point specified on the last statement is used.

EXPAND statement

The EXPAND statement is placed immediately following the INCLUDE statement. The maximum number of bytes that can be added to any indicated section is 4095.

IDENTIFY statement

An IDENTIFY statement can be continued. A whole operand must appear on a single line, and at least one operand must appear on each line of a continued statement.

Placement: The linkage editor requires that the IDENTIFY statement follow the module containing the control section to be identified or the INCLUDE statement specifying the module.

NAME statement

If a name is not specified on a NAME statement, the name TEMPNAME will be assigned to the module.

ORDER statement

If the same common area or control section is listed on more than one ORDER statement, the linkage editor uses the sequence listed on the first statement. The linkage editor ignores all subsequent occurrences of the name and the balance of

the ORDER statement on which the name appears except when the occurrence is the last operand on one ORDER statement and the first operand on the next.

REPLACE statement

Placement: If the REPLACE statement is the last control statement in the SYSLIN data set, and there are unresolved external references to be resolved from SYSLIB, the linkage editor causes the REPLACE service to operate on the first module from SYSLIB by an automatic library call.

When a control section containing unresolved external references is deleted, the unresolved references remain in the CESD.

When some but not all control sections of a separately assembled module are to be replaced, the linkage editor causes A-type address constants that refer to a deleted symbol to be incorrectly resolved unless the entry name is at the same displacement from the origin in both the old and the new control sections.

If no INCLUDE statement follows the REPLACE statement, one module might be left out during automatic library call. Message IEW0132 is issued.

Unsupported binder control statements

The following binder control statements are not supported:

- AUTOCALL
- IMPORT
- RENAME
- SETOPT

Processing and attribute options reference

The options described in Chapter 6, “Binder options reference,” on page 67 also apply to the linkage editor and batch loader except as noted here.

Supported binder options

The linkage editor and batch loader support the following binder options:

- AC
- ALIGN2
- AMODE
- CALL
- DC
- DCBS
- EP
- LET
- LIST
- LISTPRIV
- MAP
- NAME
- OL
- OVLY
- PATHMODE
- PRINT
- RES
- RMODE
- SCTR

- SIZE
- TERM
- TEST
- XCAL
- XREF

LIST: Listing control

Specify **LIST** or **NOLIST**. The form **LIST=***value* is not supported by the linkage editor and batch loader. When the LIST option is specified, the control statements are listed in either the SYSPRINT, SYSLOUT, or SYSTERM data set.

MAP and XREF

When the XREF option is specified, the linkage editor produces a cross-reference table of the output load module. The cross-reference table includes a module map; therefore, both XREF and MAP need not be specified in the same job step.

Reusability

The form **REUS(***value***)** is not supported by the linkage editor. Use the single keyword form **REUS | NOREUS | RENT | NORENT | REFR | NOREFR** to code the reusability option. See “REUS: Reusability options” on page 87 for further information on reusability attributes.

SIZE: Space specification

value1
: For the linkage editor, the minimum value is 96KB (98304 bytes) and the maximum value is 9999KB (approximately 10MB). All of this storage is below the 16 MB line.

value2
: The minimum value is the larger of 6KB (6144) or the length of the largest input text record. The maximum value is the length of the output load module plus 4096 bytes if the length of the output module is equal to or greater than 40KB.

  The storage specified by *value2* is part of the total allocation specified by *value1*.

Not-Executable attribute

Unlike the binder, the linkage editor will replace an executable module with a notexecutable version. All other conditions, such as the replace option on the NAME statement and the LET option, must allow for storing of the module.

Incompatible processing and attribute options

Some processing and attribute options are incompatible: Some options cannot be active at the same time with others. In Figure 38 on page 164, an X at an intersection marks a pair of incompatible options. When both are specified, the option that appears higher in the list is used. For example, if both **OVLY** and **RENT** are specified, the module will be in an overlay structure but is not reenterable.

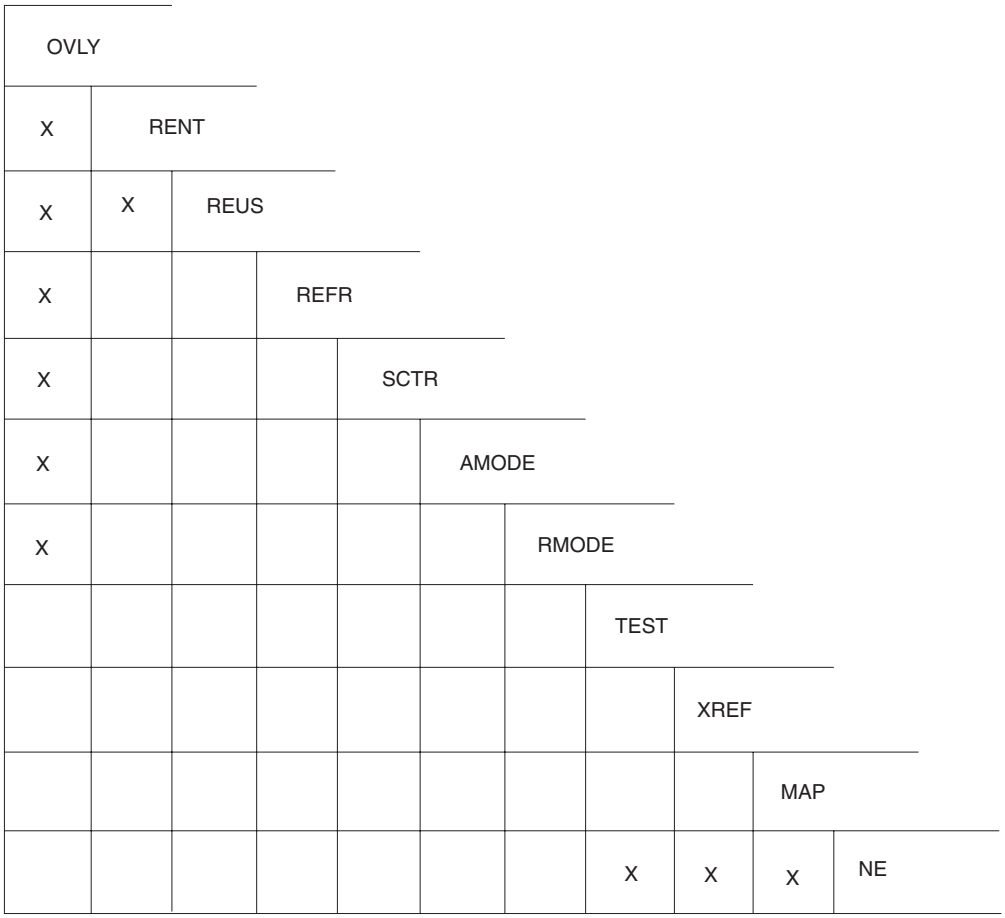


Figure 38. Incompatible processing and attribute options. Options not shown here can be specified in any combination.

Linkage editor requirements

This section describes the amount of virtual storage the linkage editor requires and its record-processing capacities.

Virtual storage requirements

The approximate minimum storage requirement and the capacity of the linkage editor program are described in Table 9. To increase the capacity for processing external symbol dictionary records, intermediate text records, relocation dictionary records, and identification records, increase *value1* or decrease *value2* of the SIZE option. Output text record length can be increased by increasing the SIZE option values, but, in no case, can the record length ever exceed the track length for the device or 32KB.

Table 9. Linkage editor capacities for minimal SIZE values (96KB, 6KB)

Function	Capacity
Virtual storage allocated	96KB
Maximum number of entries in CESD	558
Maximum number of intermediate text records	372

Table 9. Linkage editor capacities for minimal SIZE values (96KB, 6KB) (continued)

Function	Capacity
Maximum number of RLD records (relocatable address constants)	192
Maximum number of segments per program	255
Maximum number of overlay regions per program	4
Maximum blocking factor for input object modules (number of 80-column card images per physical record)	5
Maximum blocking factor for SYSPRINT output (number of 121-character logical records per physical record)	5
Output text record length, for the devices supported by this system	3KB (See Note)
Note: The maximum output text record length is achieved when *value2* of the SIZE parameter is at least twice the record length size.	

The number of overlay segments and regions that can be processed is not affected by increasing the available storage.

For the CESD, the number of entries allowed can be computed by subtracting, from the maximum number given in Table 9, one entry for each of the following:

- A ddname specified in LIBRARY statements
- A ddname specified in INCLUDE statements
- An ALIAS statement
- A symbol in REPLACE or CHANGE statements that are in the largest group of these statements preceding a single object module in the input to the linkage editor
- The segment table (SEGTAB) in an overlay program
- An entry table (ENTAB) in an overlay program.

To compute the number of intermediate text records that will be produced during processing of either program, add one record for each group of *x* bytes within each control section, where *x* is the record size for the intermediate data set. The minimum value for *x* is 1KB; a maximum is chosen depending on the amount of storage available to the linkage editor and the devices allocated for the intermediate and output data sets.

The number of intermediate text records that can be handled by a linkage editor program is less than the maximums given in Table 9 if the text of one or more control sections is not in sequence by address in the input to the linkage editor.

The total length of the data fields of the CSECT identification records associated with a load module cannot exceed 32KB. To determine the number of bytes of identification data contained in a particular load module, use the following formula:

SIZE = 269 + 16A + 31B + 2C + I(n + 6)

where:

A = The number of compilations or assemblies by a processor supporting CSECT identification that produced the object code for the module.

B = The number of preprocessor compiler compilations by a processor supporting CSECT identification that produced the object code for the module.

C = The number of control sections in the module with END statements that contain identification data.

I = The number of control sections in the module that contain user-supplied data supplied during link-editing by the optional IDENTIFY control statement.

n = The average number of characters in the data specified by IDENTIFY control statements.

Notes:

1. The size computed by the formula includes space for recording up to 19 AMASPZAP modifications. When 75% of this space has been used, a new 251-byte record is created the next time the module is reprocessed by the linkage editor.
2. To determine the approximate number of records involved, divide the computed size of the identification data by 256.

Example: A module contains 100 control sections produced by 20 unique compilations. Each control section is identified during link-editing by 8 characters of user data specified by the IDENTIFY control statement. The size of the identification data is computed as follows:

```
A = 20
I = 100
n = 8

269 + 320 + 1400 = 1989 bytes
```

If the optional user data specified on the IDENTIFY control statements is omitted, the size can be reduced considerably as shown in the following computation:

```
269 + 320 = 589 bytes
```

The maximum number of downward calls made from a segment to other segments lower in its path can never exceed 340. To compute the maximum number of downward calls allowed, subtract 12 from the SYSLMOD record size, divide the difference by 12. Examples of maximum downward calls are 84 for a SYSLMOD record size of 1024 bytes and 340 for a SYSLMOD record size of 6144 bytes.

Batch loader requirements

The batch loader can require virtual storage space for the following items:

- Batch loader code
- Data management access methods
- Buffers and tables used by the batch loader (dynamic storage)
- Loaded program (dynamic storage).

Region size includes all four of these items; the SIZE option refers to the last two items.

For the SIZE option, the minimum required virtual storage is 4KB plus the size of the loaded program. This minimum requirement grows to accommodate the extra

table entries needed by the program being loaded. For example, Fortran requires at least 3KB plus 4KB plus the size of the loaded program, and PL/I needs at least 8KB plus 4KB plus the size of the loaded program. Buffer number (BUFNO) and block size (BLKSIZE) could also increase this minimum size. Table 10 shows the appropriate storage requirements in bytes.

The maximum virtual storage that can be used is whatever virtual storage is available.

All or part of the storage required is obtained from user storage.

Table 10. Batch loader virtual storage requirements

Consideration	Approximate virtual storage requirements (in bytes)	Comments
Data Management	6KB	BSAM
Object Module Buffers and DECBs	BUFNO × (BLKSIZE + 24)	Concatenation of different BLKSIZE and BUFNO must be considered. (Minimum BUFNO=2)
Load Module Buffer and DECBs	304	
SYSTERM DCB Buffers and DECBs	312	Allocated if TERM option is specified
SYSLOUT Buffers and DECBs	BUFNO × (BLKSIZE + 24)	Buffer size rounded up to integral number of double words. (Minimum BUFNO=2)
Size of program being loaded	Program size	Program size is restricted only by available virtual storage
Each external relocation dictionary entry	8	
Each external symbol	20	
Largest ESD number	4n (n is the largest number of ESDs in any input module)	Allocated in increments of 32 entries
Fixed Loader Table Size	1260	Subtract 88 if NOPRINT is specified
Condensed Symbol Table	12n (n is the total number of control sections and common areas in the loaded program)	Built only if you invoke the binder under TSO, and space is available.
System Requirements	4000	

Interpreting linkage editor output

Diagnostic output

Diagnostic information is written to the diagnostic output data set that is defined by a SYSPRINT DD statement. The diagnostic report consists of a header and linkage editor messages. There are two types of messages: module disposition, which are described in “Module disposition messages” on page 168, and error/warning messages, which are described in *z/OS MVS System Messages, Vol 8 (IEF-IGD)*.

Output listing header

The output listing header includes:

- The time, day of the week, and date that the link-edit job was run.

- The job name you have specified and the job step name.
- The invocation parameters you have specified.
- The amount of working storage used and the output buffer size. These two values are shown as:

  ACTUAL SIZE=(*value1*,*value2*)

  where:

  value1 = The actual amount of working storage that the linkage editor used and not the value you requested.

  value2 = The actual output buffer size and not the value you requested.
- The name of the SYSLMOD data set and its volume.

Invalid options and attributes are replaced by INVALID in the output listing header. If incompatible attributes are specified, additional messages are generated.

Module disposition messages

Module disposition messages are generated for each load module produced. There are two groups of messages. The first group of disposition messages describes the handling of the load module. These messages are:

- *member name* ADDED AND HAS AMODE *addressing mode*
- *member name* REPLACED AND HAS AMODE *addressing mode*
- *member name* DID NOT PREVIOUSLY EXIST BUT WAS ADDED AND HAS AMODE *addressing mode*

  In this case, the replacement function was specified, but the member did not exist in the data set; the module is added to the data set using the member name given.
- *alias name* IS AN ALIAS AND HAS AMODE *addressing mode*
- MODULE HAS BEEN MARKED NOT EXECUTABLE.
- LOAD MODULE HAS RMODE *residence mode*
- AUTHORIZATION CODE IS *authorization code*.

The second group of module disposition messages is generated when reenterable (RENT), reusable (REUS), or refreshable (REFR) linkage editor options have been specified for the module. A message indicates whether the load module has been marked reenterable or not reenterable, reusable or not reusable, refreshable or not refreshable, depending on the option or options used.

The RENT/REUS/REFR message consists of MODULE HAS BEEN MARKED, followed by the attributes assigned. The following messages are examples of some possible combinations:

- MODULE HAS BEEN MARKED REFRESHABLE.
- MODULE HAS BEEN MARKED NOT REFRESHABLE.
- MODULE HAS BEEN MARKED REUSABLE AND NOT REFRESHABLE.
- MODULE HAS BEEN MARKED REUSABLE AND REFRESHABLE.

When an error causes the linkage editor to mark a module not executable, only the MODULE HAS BEEN MARKED NOT EXECUTABLE message appears; no attribute messages are generated.

Error/Warning messages

Certain conditions that are present when a module is being processed can cause error or warning messages to be printed. These messages contain a message code and message text. If an error is encountered during processing, the message code for that error is printed with the applicable symbol or record in error. After processing is completed, the diagnostic message associated with that code is printed.

The error warning messages have the following format:

IEW0mms message text

where:

IEW0 Indicates a linkage editor message

mm Is the message number

s Is the severity code, and can be one of the following values:

- **1** Indicates a condition that might cause an error during execution of the output module. A module map or cross-reference table is produced if specified by you. The output module is marked executable.
- **2** Indicates an error that could make execution of the output module impossible. Processing continues. When possible, a module map or a cross-reference table is produced if specified by you. The output module is marked not executable, unless the LET option is specified on the EXEC statement.
- **3** Indicates an error that will make execution of the output module impossible. Processing continues. When possible, a module map or a cross-reference table is produced if specified by you. The output module is marked not executable.
- **4** Indicates an error condition from which no recovery is possible. Processing terminates. The only output is diagnostic messages.

Note: A special severity code of zero is generated for each control statement printed as a result of the LIST option. Severity zero does not indicate an error warning condition.

The highest severity code encountered during processing is multiplied by 4 to create a return code that is placed in register 15 at the end of processing. This return code can be tested to determine whether processing is to continue.

message text contains combinations of the following:

- The message classification (either error or warning)
- Cause of error
- Identification of the symbol, segment number (when in overlay), or input item to which the message applies
- Instructions to the programmer
- Action taken by the linkage editor.

z/OS MVS System Messages, Vol 8 (IEF-IGD) contains a complete list of the linkage editor error and warning messages.

Sample diagnostic output

Figure 39 shows the format of the diagnostic output for the linkage editor. No optional output was requested other than the list of control statements.

```
A  z/OS V1 R3  LINKAGE EDITOR 16:52:40  MON  JANUARY 28,2002
    JOB MAINRUN     STEP LINKEDIT
    INVOCATION PARAMETERS  -  LET,NCAL,XREF,LIST
    ACTUAL SIZE=(317440,86016)
    OUTPUT DATA SET USER.LOADLIB IS ON VOLUME SYS086

B   IEW0000     NAME BBBBBBBB(R)
       IEW0461 CCCCCCCC
       IEW0461 BASEDUMP

C   ** BBBBBBBB ADDED AND HAS AMODE 24
    ** LOAD MODULE HAS RMODE 24
    ** AUTHORIZATION CODE IS           0.
                                              DIAGNOSTIC MESSAGE DIRECTORY

D   IEW0461 WARNING - SYMBOL PRINTED IS AN UNRESOLVED EXTERNAL REFERENCE, NCAL WAS SPECIFIED
```

Figure 39. Diagnostic messages issued by the linkage editor

The figures on the left side of Figure 39 indicate the portion of the diagnostic output being described.

A Is the output listing header. It contains a time and date stamp, invocation parameters specified by you, storage and buffer sizes, and the name of the SYSLMOD data set and its volume. In this example, MAINRUN and LINKEDIT are the user-specified job name and step name, respectively.

B Is a list of control statements used (IEW0000) and the message codes (IEW0461) for error/warning conditions discovered during processing. For error/warning message codes, the symbol in error, if necessary, is also listed (CCCCCCCC and BASEDUMP).

C Is a module disposition message indicating that the output module (BBBBBBBB) has been added to the output module data set. The addressing and residency modes and the module authorization code are listed.

D Is the diagnostic message directory that contains the text of the error codes listed in item **B**.

Optional output

In addition to error/warning and disposition messages, the linkage editor can produce diagnostic output as requested by you. This optional output includes a control statement listing, a module map, and a cross-reference table.

Control statement listing

If the LIST option is specified on the EXEC statement, a listing of all linkage editor control statements is produced. For each control statement, the listing contains a special message code, IEW0000, followed by the control statement. Item **B** in Figure 39 contains an example of a control statement listing.

Module map

If the MAP option is specified on the EXEC statement, a module map of the output load module is produced. The module map shows all control sections in the output module and all entry names in each control section. Named common areas are listed as control sections.

For each control section, the module map indicates its origin (relative to zero) and length in bytes (in hexadecimal notation). For each entry name in each control section, the module map indicates the location where the name is defined. These locations are also relative to zero.

If the module is not in an overlay structure, the control sections are arranged in ascending order according to their origins. An entry name is listed with the control section in which it is defined.

If the module is an overlay structure, the control sections are arranged by segment. The segments are listed as they appear in the overlay structure, top to bottom, left to right, and region by region. Within each segment, the control sections and their corresponding entry names are listed in ascending order according to their assigned origins. The number of the segment in which they appear is also listed.

In any module map, the following are identified by a dollar sign:
- Blank common area
- Private code (unnamed control section)
- For overlay programs, the segment table and each entry table.

When the load module processed by the linkage editor does not have an origin of zero, the linkage editor generates a one-byte private code (unnamed control section) as the first text record. This private code is deleted in any subsequent reprocessing of the load module by the linkage editor.

Each control section that is obtained from a call library during automatic library call is identified by an asterisk after the control section name.

At the end of the module map is the entry address, that is, the relative address of the main entry point. The entry address is followed by the total length of the module in bytes; in the case of an overlay module, the length is that of the longest path. Pseudoregisters, if used, also appear at the end of the module map; the name, length, and displacement of each pseudoregister are given.

Figure 40 on page 172 contains a module map and cross-reference listing with four control sections. There are three named control sections (ABC00, ABCSUB1, and ABCSUB2) and one unnamed control section (designated by $PRIVATE). Control sections ABCSUB1 and ABCSUB2 were obtained from a call library. Control section ABCSUB1 also has two additional entry points. The entry point for control section ABCSUB2 is named ABCENT2.

```
                                        CROSS REFERENCE TABLE

  CONTROL   SECTION           ENTRY
    NAME    ORIGIN  LENGTH    NAME     LOCATION   NAME     LOCATION   NAME   LOCATION  NAME   LOCATION
  $PRIVATE       0       8
  ABC00         08    1004
  ABCSUB1*    100C      DE
                            ABCSUB1    100C     ABCSUB1A    1016    ABCHLP1   108E
  ABCSUB2*    10E8     767
                            ABCENT2    10E8

  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION
      31F            ABCSUB1         ABCSUB1          325            ABCSUB1A        ABCSUB1
      354            ABCENT2         ABCSUB2          360            ABCHLP1         ABCSUB1
      364            ABCSUB1A        ABCSUB1
 ENTRY ADDRESS       08
 TOTAL LENGTH      1850
```

Figure 40. Linkage editor module map and cross-reference table

Cross-reference table

If the XREF option is specified on the EXEC statement, a cross-reference table is produced. The cross-reference table consists of a module map and a list of cross-references for each control section. Each address constant that refers to a symbol defined in another control section is listed with its assigned location, the symbol referred to, and the name of the control section in which the symbol is defined. When control sections are compiled together, and simple address constants are used to refer from one control section to another (instead of using external symbols and entry names), the control section name is listed as the symbol referred to.

For overlay programs, this information is provided for each segment; the number of the segment in which the symbol is defined is also provided.

If a symbol is unresolved after processing by the linkage editor, it is identified by $UNRESOLVED in the list. However, if an unresolved symbol is marked by the never-call function (as specified on a LIBRARY control statement), it is identified by $NEVER-CALL. If an unresolved symbol is a weak external reference, it is identified by $UNRESOLVED(W).

Figure 40 on page 172 includes a cross-reference table of the address constants in program ABC00.

Linkage editor return codes

Control is passed to the linkage editor as a job step when the linkage editor is specified on an EXEC job control statement in the input stream. When the job step is completed, the linkage editor passes a return code to the control program.

The return code reflects the highest severity code recorded in any iteration of the linkage editor within that job step. The highest severity code encountered during processing is multiplied by 4 to create the return code; this code is placed into register 15 at the end of linkage editor processing. Table 11 on page 173 contains the return codes, the corresponding severity code, and a description of each.

Table 11. Linkage editor return codes

Return code	Severity code	Description
0	0	Normal conclusion
4	1	Warning messages have been listed; execution should be successful.
8	2	Error messages have been listed; execution might fail. The module is marked not executable unless the LET option is specified.
12	3	Severe errors have occurred; execution is impossible.
16	4	Terminal errors have occurred; the processing has terminated.

Interpreting batch loader output

The batch loader output consists of a collection of diagnostic and error messages and an optional storage map of the loaded program. The output is produced in the data set defined by the SYSLOUT DD and SYSTERM DD statements. If these statements are omitted, no output is produced.

SYSLOUT output includes a heading, and the list of options and defaults requested through the PARM field of the EXEC statement. The SIZE stated is the size obtained, and not necessarily the size requested in the PARM field. Error messages are written when the errors are detected. After processing is complete, an explanation of the error is written. *z/OS MVS System Messages, Vol 8 (IEF-IGD)* lists the batch loader error messages.

SYSTERM output includes only numbered warning and error messages. These messages are written when the errors are detected. After processing is complete, an explanation of each error is written.

The storage map includes the name and absolute address of each control section and entry point defined in the loaded program. Each map entry marked with an asterisk (*) comes from the data set specified on the SYSLIB DD statement. Two asterisks (**) indicate the entry was found in the link pack area; three asterisks (***) indicate the entry comes from text that was preloaded by a compiler. The TYPE column indicates what each entry on the map is used for: SD=control section, LR=label reference, and PR=pseudoregister.

The map is written as the input to the batch loader is processed, so all map entries appear in the same sequence in which the input ESD items are defined. The total size and storage extent of the loaded program are also included. For PL/I programs, a list is written showing pseudoregisters with their addresses assigned relative to zero. Figure 41 on page 174 shows an example of a module map. The batch loader issues an informational message when the loaded program terminates abnormally.

```
 NAME   TYPE ADDR   NAME    TYPE ADDR   NAME     TYPE ADDR   NAME    TYPE ADDR  NAME   TYPE ADDR

SAMPL2B   SD 161E0  SAMPL2BA  SD 16EC8  IHEMAIN   SD  17CF8  IHENTRY   SD 17D00 IHESPRT  SD 17D10
SYSIN     SD 17D48  IHEVQC  * SD 17D80  IHEVQCA * LR  17D80  IHEVQB  * SD 17FD8 IHEVQBA* LT 17FD8
IHEDIA  * SD 183C0  IHEDIAA * LR 183C0  IHEIAB  * LR  183C2  IHEVPE  * SD 18608 IHEVPEA* LR 18608
IHEVPA  * SD 18870  IHEVPAA * LR 18870  IHEVFC  * SD  189D0  IHEVFCA * LR 189D0 IHEVPC * SD 189F8
IHEVPCA * LR 189F8  IHEVFE  * SD 18BE8  IHEVFEA * LR  18BE8  IHEVSC  * SD 18C08 IHEVSCA* LR 18C08
IHEDNC  * SD 18CB8  IHEDNCA * LR 18CB8  IHEDOA  * SD  18F30  IHEDOAA * LR 18F30 IHEDOAB* LR 18F32
IHEDMA  * SD 19010  IHEDMAA * LR 19010  IHEVFD  * SD  19108  IHEVFDA * LR 19108 IHEVFA * SD 19160
IHEVFAA * LR 19160  IHEVPB  * SD 19248  IHEVPBA * LR  19248  IHEXIS  * SD 193F0 IHEXISO* LR 193F0
IHEIOB  * SD 19488  IHEIOBA * LR 19488  IHEIOBB * LR  19490  IHEIOBC * LR 19498 IHEIOBD* LR 194A0
IHESARC * LR 1A9CB  IHESADD * LR 1A9DE  IHESAFF * LR  1AA18  IHEPRT  * SD 1AB70 IHEPRTA* LR 1AB70
IHEBEGA * LR 1AE28  IHEERR  * SD 1AE68  IHEERRD * LR  1AE68  IHEERRC * LR 1AE68 IHEERRB* LR 1AE7C
IHEERRA * LR 1AE68  IHEERRE * LR 1B4E2  IHEIOF  * SD  1B580  IHEIOFR * LR 1B580 IHEIOFA* LR 1B582
IHEITAZ * LR 1B81E  IHEITAX * LR 18B2A  IHEITAA * LR  1B83E  IHEDCNR * SD 1B680 IHEDCNA* LR 1B860
IHEDCNB * LR 1B862  IHEIOD  * SD 1BA50  IHEIODG * LR  1BA50  IHEIODP * LR 1BA52 IHEIODT* LR 1BB4A
IHEVTB  * SD 1BCF0  IHEVTBA * LR 1BCF0  IHEVQA  * SD  1BD78  IHEVQAA * LR 1BD78


IHEQINV   PR    00  IHEGERR   PR     4  SAMPL2BB  PR      8  SAMPL2BC  PR     C IHEQSPR  PR    10
SYSIN     PR    14  IHEQLSA   PR    18  IHEQLWO   PR     1C  IHEQLW1   PR    20 IHEQLW2  PR    24
IHEQLW3   PR    28  IHEQLW4   PR    2C  IHEQLWE   PR     30  IHEQLCA   PR    34 IHEQVDA  PR    38
IHEQFVD   PR    3C  IHEQFCL   PR    40  IHEQFOP   PR     48  IHEQADC   PR    4C IHEQXLV  PR    50
IHEQEVT   PR    58  IHEQSLA   PR    60  IHEQSAR   PR     64  IHEQLWF   PR    68 IHEQRTC  PR    6C
IHEQDFC   PR    70

IEW1001    IHEUPBA
IEW1001    IHEUPAA
IEW1001    IHETERA
IEW1001    IHEM91C
IEW1001    IHEM91B
IEW1001    IHEM91A
IEW1001    IHEDDOD
IEW1001    IHEVPFA
IEW1001    IHEVPDA
IEW1001    IHEDBNA
IEW1001    IHEVSFA
IEW1001    IHEVSBA
IEW1001    IHEVCAA
IEW1001    IHEVSAA
IEW1001    IHEDNBA
IEW1001    IHEUPBB
IEW1001    IHEUPAB
IEW1001    IHEVSEB

  TOTAL LENGTH   5068
  ENTRY ADDRESS  17D00

IEW1001  WARNING — UNRESOLVED EXTERNAL REFERENCE (NOCALL SPECIFIED)
```

Figure 41. Batch loader module map

Batch loader return codes

The return code of a loader step is determined by the return codes resulting from batch loader processing *and* from loaded program processing.

The return code indicates whether errors occurred during the execution of the loader or of the loaded program. The return code can be tested through the COND parameter of the JOB statement specified for this job or the COND parameter of the EXEC statement specified in any succeeding job step (see *z/OS MVS JCL User's Guide*). Table 12 on page 175 shows the return codes.

Note: Error diagnostics (SYSLOUT or SYSTERM data set, or both) for the loader will show the severity of errors found by the loader.

Table 12. Batch loader return codes

Code returned to caller	Loader return code	Program return code	Description
0	0	0	Program loaded successfully, and execution of the loaded program was successful.
0	4	0	The batch loader found a condition that might cause an error during execution, but no error occurred during execution of the loaded program.
0	8(LET)	4	The batch loader found a condition that might cause an error during execution, but no error occurred during execution of the loaded program.
4	0	4	Program loaded successfully, and an error occurred during execution of the loaded program.
4	4	4	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
4	8(LET)	4	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
8	0	8	Program loaded successfully, and an error occurred during execution of the loaded program.
8	4	8	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
8	8(LET)	8	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
8	8		The batch loader found a condition that could make execution impossible. The loaded program was not executed.
12	0	12	Program loaded successfully, and an error occurred during execution of the loaded program.
12	4	12	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
12	8(LET)	12	The batch loader found a condition that might cause an error during execution, and an error did occur during execution of the loaded program.
12	12		The batch loader could not load the program successfully; execution impossible.
16	0	16	Program loaded successfully, and the loaded program found a terminating error.
16	4	16	The batch loader found a condition that might cause an error during execution, and a terminating error was found during execution of the loaded program.
16	8(LET)	16	The batch loader found a condition that might cause an error during execution, and a terminating error was found during execution of the loaded program.
16	16		The batch loader could not load program; execution impossible.

Loader serviceability aids

The following are serviceability aids provided in the loader:

- The control section, HEWLDDEF, contains the loader option default values. It is resident in load module HEWLOADR.

- A storage dump will typically produce information on the nature of the error. Register 11 will contain a pointer to HEWLDCOM, and register 12 will contain the base register associated with the CSECT in control.
- All nine save areas are forward and backward chained. Lower-level save areas will be printed. A hexadecimal “FF” in word 4 of the save area indicates that the routine represented by the save area has returned control. At the entry point to each module, register 13 contains the save area address and register 14 contains the return address.
- Input/output control information is contained in the loader communication area. This information consists of the DECB address, the buffer locations, the block size, the logical record length, the blocking factor, the number of records left in the buffer, the address of the current record, and the associated switches.
- Appropriate diagnostic messages are produced when an error has been detected. The message has a specific number and, where appropriate, lists the data in error. The message number and text are listed by HEWLLIBR at the end of loading.
- The loader uses the SYNADAF macro to obtain information regarding permanent I/O errors, and lists the information on the SYSLOUT data set.

In addition to the above, you may choose to use the AMBLIST service aid to print the contents of the input object modules, load modules, or program objects. See “The AMBLIST service aid” on page 153.

Appendix B. Summary of Program Management user considerations

Migrating from the linkage editor to the binder

The binder has replaced the linkage editor and batch loader programs as the system default linker and linker-loader, respectively. Except as noted in this section, the binder assumes all of the functions of the other two linking programs. Invoking any of the common linkage editor or batch loader entry points, such as IEWL, HEWL, LINK, and LOADER, will result in execution of the binder.

While the binder includes all of the functions of the linkage editor and batch loader, it is not fully compatible with those programs. It was developed in response to many customer, vendor and internal requirements requesting relief from various restrictions and processing anomalies in the older programs. The binder attempts to satisfy many of those requirements as well as provide a consistent processing model. As a result it provides a set of externals, which is similar but not identical to the linkage editor and batch loader externals.

The linkage editor and batch loader are also available in z/OS. There are no plans to withdraw either of those programs at this time, but all users are encouraged to begin using the binder as early as possible. In cases where the binder appears unsuitable for a specific application, the older programs are unchanged and can be invoked by entry names HEWLKED or HEWLF064 (linkage editor) or HEWLDIA (batch loader). Note, however, that all future enhancements will be made to the binder and loader exclusively. Other IBM products might have dependencies on functions provided only by these components.

Installations that share DASD volumes between systems at different system levels must ensure that the level of the binder being used matches the level of the system it is running on. In addition, users must be sensitive to functional and format differences in binder processing and output if sharing modules between different releases of z/OS and OS/390.

SMP/E precautions

When using the System Modification Program Extended (SMP/E) for software installation, the system programmer should be aware of the following: In z/OS, the binder is the default linker program invoked by SMP/E. Because the binder handles some error conditions differently than did the linkage editor, it is possible that certain error conditions might go unnoticed during the installation process.

Binder-detected errors that could cause the linked program to fail during execution are reported with an error message and a return code 8 being passed back to SMP/E. In cases where conflicting input might or might not represent an error, a warning message and return 4 will be provided. Since SMP/E recommends that users specify a maximum return code of 8 in the linkedit utility entry in the global zone, conflicting or incomplete input to the binder might go undetected during SMP/E APPLY processing. Because the binder's default action in these error situations might be different from that of the linkage editor, the results of the installation might be different with the binder. System programmers are strongly encouraged to check all severity 8 error messages from the binder.

Storage considerations using the binder

The binder requires a larger region than does the linkage editor. This is because the binder has relaxed most of the restrictions inherent in the linkage editor, replacing fixed-length tables with open-ended lists that require more storage. In addition, the binder does not use a DASD work file for spilling module data when processor storage has been exhausted, as does the linkage editor. The SYSUT1 DD statement is ignored. Instead, it uses primary or data space storage for all module data. Because of the free-form design of the binder's internal data structures and the number of controlling factors involved, it is not possible to accurately predict binder storage requirements.

It is recommended that the binder be given a region of at least 2 MB, larger for very large modules or modules consisting of a large number of CSECTs, external names or address constants. Most binder working storage will be obtained from above 16MB, if sufficient space is available in the extended private area. Installations that restrict the extended region size default through use of the IEFUSI installation exit might force the binder to obtain its storage from the private area below 16MB. In such cases, the binder user might be forced to specify a very large region size, such as 16M, in order to obtain sufficient storage in the extended region.

Message IEW2971T can be issued for a very large module. The solution is for the system programmer to change the behavior of the installation's IEFUSI exit. For proper Binder operation when linking very large modules (especially C++ or Java modules) the exit should permit multi-megabyte data spaces. In exceptional situations, such as when installing a large product using SMP/E, IEFUSI algorithms could be temporarily modified to allow larger data spaces.

It is also recommended that binder users do not specify SIZE or WKSPACE as a binder execution parameter, unless the binder will be co-resident with another processing program. Either of these options will limit the amount of storage available to the binder and, if insufficient, might cause the binder to fail with an out-of-storage condition. The problem is aggravated if insufficient extended region is available and all binder working storage is forced below 16MB.

Error handling in the binder

The binder is less tolerant of errors and inconsistencies in its input than was the linkage editor. Error conditions were frequently ignored or overridden by the linkage editor, which might or might not be what the user intended. Often such errors and the resulting system action went unreported.

The binder attempts to diagnose all such error conditions and take a course of action that is consistent with its general processing model. Input modules and other files that are inaccessible or are in an incorrect format will generally be omitted. Control statements and parameters containing invalid syntax or data will also be discarded. All such errors will result in an error message and a return code 8. Conflicting and inconsistent specifications and data might result in either a warning (severity 4) or error (severity 8) being issued, depending on the seriousness of the condition and the likelihood of program failure during execution.

As a result, the binder issues many more messages than did the linkage editor. The binder contains nearly four times the number of unique error messages as did the linkage editor and batch loader combined, in an effort to more accurately diagnose error conditions.

Changes and extensions in output using the binder

The binder provides significant extensions in output, such as, error messages, output listings, information included, for example:

- Messages are more numerous, accurate and informative. (In fact, users can choose to use MSGLEVEL to suppress some messages.)
- Output listings provide information about the binding job, more alias information, and operational and summary data.
- Output listings include the binder release level, processing options and program attributes.
- Default for output listings is LIST=SUMMARY. This will cause the following (more than for the linkage editor) to be printed:
  - Target library (SYSLMOD) description
  - Processing options
  - Date/time of SAVE
  - Module attributes (that are stored in directory)
  - Entry points
- By specifying the MAP option, output listings will also include the source of each CSECT in the module, specifically the ddname, member name, concatenation number, and a cross-reference table of ddname to dsname.

If you do not want to receive all of this output, several options are available to limit the amount of printed material produced during binder processing:

- The LIST option can be used to limit the volume of automatic printed output, such as the echoing of control statements and the generation of the processing summary report.
- Not specifying the MAP and XREF options will significantly reduce the amount of printed output generated for those reports.
- Specifying MSGLEVEL will allow you to suppress messages below a certain severity level.
- Specifying the suboption NOIMPort on the LIST option will suppress the echoing of import statements for DLLs.

Note: Remember that limiting binder printed output in any of these ways might hide problems in your module.

Binder control statements and options

Note: Certain processing differences must be considered when migrating from the linkage editor to the binder. Subtle differences in the way control statements and options are processed might affect the resultant load module or program object. Differences between PDS and PDSE libraries might also affect the results. Some of these differences are described below.

Several of the binder control statements and processing options have interrelated functions. The binder attempts to process both in a consistent way, even though the processing can deviate from that of the linkage editor. Toward this end, the following rules are observed when processing data from all sources (included modules, control statements, specified options or API function calls):

- Control statements always override the corresponding batch parameters. The scope of the control statement is the module in process.

- Batch parameters, including those specified on the STARTDialog function call, always override the input module, such as ESD data. The scope of the batch parameters is the entire binder invocation or dialog.
- Module data always prevails over binder default values.
- If duplicate specifications are encountered, the most recent specification will prevail. That is, the binder processes the last occurrence of control statements and options. (The linkage editor processes the first or last depending on option.)
  - When there are multiple ENTRY statements (there should not be), the binder will process the last ENTRY statement whereas the linkage editor will process the first ENTRY statement. This could result in execution errors if conflicting ENTRY statements are present.
  - Control statements and parameter strings are always processed in a left-to-right sequence. Function calls are processed in the order received.
- Control statements and parameters containing invalid syntax, keywords or values, will be discarded and reported as errors.

Binder processing differences from the linkage editor

The binder behavior might be different from the linkage editor in some significant ways:

- The linkage editor ignored data it didn't recognize or couldn't process. The binder also discards nonprocessable input, but diagnoses the error with a message and nonzero return code.
- The linkage editor accepted the ***first ENTRY control statement*** encountered, whereas the binder accepts the last. This could result in execution errors if the multiple statements specify conflicting entry points.
- Unlike the linkage editor, ***explicit AMODE and RMODE specifications*** during binder processing always override the corresponding attributes in the ESD of included modules. A new MIN value has been provided for AMODE to allow ESD influence over the results. RMODE(MIN) is the default and can not be specified.
  - AMODE and RMODE are treated as independent options until they are needed during binder processing. The linkage editor processes them as a pair. If only one of the pair is specified on either the parm string or a control statement, the other will be set depending on the one specified. If neither option is specified or both are specified, the binder will behave like the linkage editor. If only one is specified, the results might be different.
  - Many object modules, especially assembler programs and programs written for older compilers, indicate AMODE(24) or RMODE(24) in their ESD records. Overriding these values at bind time will produce warning messages IEW2646I and IEW2651I, one per section in error. The linkage editor ignored the condition but the binder assumes that a valid error condition might exist. By specifying the binder option COMPAT=LKED (see below), you can force the binder to suppress these messages and leave the return code unchanged.
- ***Reusability (REUS, RENT and REFR)*** is handled differently by the binder. While the linkage editor processes the attributes independently, the binder stores them as a single value. The binder assumes that reenterable programs are also serially reusable, and the refreshable programs are also reenterable. This should not cause any processing difficulties.
  - The binder was designed to always accept an explicit override of a module attribute, whereas the linkage editor sometimes does not. For example, although the JCL can specify RENT in the parm list, when one CSECT being bound into a load module is reusable and the rest are reentrant, the linkage

editor ignores the external parameter and assigns the module as reusable. The binder will allow the explicit override of RENT on the JCL to take priority.

- Since the release of the binder, customer feedback indicated there has been some dependence on the internals of the linkage editor processing in two areas: module attribute defaulting and AMODE/RMODE consistency.
  - Many job streams specify RENT with the expectation that the linkage editor would look at all the pieces and assign the highest level reusability it could, for example, the customer expected the linkage editor to override any external parameters.
  - Many programs in the field continue to be bound with inconsistent AMODE/RMODE specifications that are known and ignored by the linkage editor.

  As a result, an ***option (COMPAT=LKED)*** was added to the binder. When this is specified in the JCL the binder will behave like the linkage editor in the following ways:
  - The binder will ignore externally specified module reusability attributes if any of the included load modules or program objects are of lesser reusability. A summary message is produced to show that the overall reusability of the module was downgraded.
  - AMODE/RMODE conflict messages (IEW2646I, IEW2651I) will not be issued by the binder when conditions such as AMODE ANY modules are combined with AMODE 24 modules.

  Note: It is essential that binder messages regarding reusability, AMODE and RMODE be analyzed. The appropriate action in all such cases is to correct the input, and perhaps to rebind the program if the attributes displayed in the binder Processing Summary are incorrect.
- The batch loader (HEWLDIA) can be used to load an in-storage object module. While this function is not supported by the binder, the binder will invoke the batch loader transparently when this interface is invoked. Applications that continue to use this interface cannot use any new functions provided by the binder. This support is limited and provided for compatibility only.

Other binder processing differences

Some binder processes that differ from the linkage editor are not directly related to binder input. These are affected by environmental differences, binder capacities and possible error conditions detected during prior processing. In general, they are not directly controllable by binder specifications and should be considered unpredictable.

- The ***order of modules*** included during autocall processing is not specifiable by the user and should therefore be considered unpredictable. Due to different autocall algorithms in the two programs, the sequence of includes will be different in the binder than it was in the linkage editor. If this sequence is important, you should provide INCLUDE control statements in the input stream. (Be aware that this only controls the order in which Csects are brought into storage by the binder. It does NOT control the final order of the Csects in the load module or program object. That is controlled by the ORDER control statement.)
- The binder handles ***nested INCLUDEs*** differently. It does not ignore all text following the nested INCLUDE as does the linkage editor.
- Specifying ***uninitialized space*** in your source program and assuming it will be initialized might provide unpredictable results during execution. Both the binder and the linkage editor fill part or all of such data areas with binary zeros, but their

algorithms are not the same. In addition, these algorithms are dependent on a number of environmental factors such as the block size and the amount of space remaining on a track.

  - If the program is sensitive to the initial values stored in large data areas, the programmer must ensure the storage is properly initialized, either at compile time or at program initialization time.
  - You can cause the binder to initialize all uninitialized areas in a PM2 or later format program object by specifying the FILL option. FILL allows you to initialize all uninitialized areas of the module and to specify the byte used for initialization. FILL cannot be used for a PM1-format program object.

- The binder will not, by default, replace an executable program with a ***nonexecutable program***. This is a departure from linkage editor processing, where the new module would replace an existing module of the same name regardless of the executability of either module. You might cause the binder to save a nonexecutable module by specifying the STORENX option in the binder's PARM field.
- The binder will ***not save an alias or alternate entry point name*** if it is the primary name of an existing member in the library. Like the linkage editor, if replace (R) has been specified on the NAME control statement and the binder discovers that the name is an alias of another member in the library, that alias will be "stolen" for the new module (load module or program object). Unlike the linkage editor, however, if the binder discovers that the alias name already exists in the library as a primary (member) name, the alias will not be stored.

  Note: This design alternative was chosen to prevent users from inadvertently specifying as an alias the name of an existing module, thereby destroying the existing module and possibly creating an unrecoverable situation in the library.
- The binder ***bypasses LLA*** when retrieving a directory entry from a PDSE or PDS during INCLUDE processing. The linkage editor first tries to obtain its directory entries from LLA. This means that if the module was modified and not refreshed in LLA, the linkage editor would not get the latest version of the module to process. The binder always gets the latest version by obtaining the directory entry directly from the library directory on DASD.
- Unlike load modules, program objects ***cannot be zapped in place***, that is, a new program object is created in the PDSE and the old one is deleted (after all connections to it are released). This means that LLA will continue to keep the old connection and will not see the modification unless that program is explicitly refreshed.
- ***Other binder improvements***:
  - There can be up to 10 temporary modules (TEMPNAM0, TEMPNAM1, ...).
  - PDSEs and PDS's can be mixed in the concatenation. Unlike the linkage editor, the binder supports SYSLIB and SYSLIN concatenation of object files with program libraries (both PDS's and PDSEs).
  - The binder allows mixed case input (190 character set) specified with the option, CASE.
  - Most of the binder resides above the-16 Mb line in ELPA. It runs in problem program state, user key.

Migrating from load modules to program objects

This section contains information for migrating from load modules to program objects.

What should be converted to program objects?

Following are considerations in determining whether or not to migrate to program objects:

- The only system library which supports program objects is SYS1.LINKLIB (plus all libraries in Linklist concatenation). SYS1.LPALIB, SYS1.NUCLEUS, and SYS1.SVCLIB are opened and accessed during IPL before the PDSE support is established and therefore can not be PDSEs.
  - However, it is possible to put program objects into LPA using the Dynamic LPA functions. This function opens the program libraries to be included dynamically after the system has been initialized, thus allowing PDSE participation. The program objects can be in any user-specified authorized PDSE program library.
- Program objects have the same restrictions as do data members in PDSEs. They cannot be accessed using EXCP, nor can there be any TTR calculations done against them. Programs requiring this access should not be converted.
- Program objects will occupy more ***space on DASD*** than did their load module counterparts. In load module format, large uninitialized areas of the program were represented by gaps in the program text; in the PM1 program object format those gaps are filled with binary zeros and written out to disk. However, gaps are reinstated in program objects in PM2 format and later. They will still take more space on DASD then load modules for several reasons. First, program objects are formatted on 4K boundaries with the minimum size being 4K, and the algorithm for compacting uninitialized space differs from that used by the linkage editor. Also, additional information is saved in program objects to allow faster loading, and to enable rebinding of C-type modules (formerly the LE prelinker discarded the rebinding information when producing its output object module).
- If new program object features are exploited, such as a length greater than 16 megabytes, or more than 32767 external names, greater than 8-byte names, multiple classes, multiparts, split-modes, or deferred classes, the program object ***cannot*** be converted back to a load module.
- PDSE program libraries can take advantage of the PDSE cross-system sharing support offered in z/OS.
- As discussed earlier, special attention must also be given to mixing specific levels of the program object with different z/OS releases.
- Only program objects can reside in z/OS UNIX files. Load modules are not supported.

Converting load modules to program objects

Once the environment has been established, program objects can be created. The data class definitions for PDSEs and the JCL/catalog procedures can be used to provide implicit migration. Various utilities can also be used to migrate modules explicitly. These include:

- IEBCOPY: can copy either single programs or entire libraries between PDS's and PDSEs. The binder is invoked to do the conversion.
- DFDSS: provides the means for migrating one or a collection of load libraries. Conversion is only done on a COPY operation, not on a DUMP/RESTORE.
- Binder: can be invoked to rebind modules for the purpose of migrating/converting them.
- OGETX can be used to copy load modules from a PDS library to z/OS UNIX files.

Compatibility of program object formats

- ***Downward Compatibility***: The default program object format is the earliest which will support the function requested by the contents of the input modules and the processing directives.
- ***Upward Compatibility***: All earlier PM functions, interfaces, formats and user job streams should work compatibly with the current release. There will be some changes in report formats and messages, where changes are necessary for this new function.
- Only PM1 format program objects support overlay format. The binder will automatically produce a PM1 version of the program object if overlay is requested and the SYSLMOD data set is a PDSE.
- During API processing for “intent access” the module will be saved in the same format it had on input if followed by a copy operation. During API processing for “intent bind” (and both libraries are PDSEs), the module will be saved in the lowest format program object which will support the requested functions unless overridden with the COMPAT option.
- If the user specifies a COMPAT value and attempts to use functions not supported by that level, the save will fail with RC=12.

Utilities, components and products that support program objects

The following is a partial list of components and products that support program objects:

- Program objects are supported by the following DFSMS utilities/services:
  - IEBCOPY
  - IEBCOMPR
  - IEHLIST
  - IEHPROGM
  - AMBLIST
  - AMASPZAP
  - TSO LOAD/GO Prompter
- Program objects are not supported by the following DFSMS utilities:
  - IEHMOVE
  - IEBDG
  - IEBGENER
  - IEBPTPCH
- ***DFdss*** support includes:
  - DUMP and RESTORE of PDSE Program Libraries, but without conversion, for example, a dumped PDSE Program Library can not be restored to a PDS.
  - COPY between PDSE and PDS Program Libraries. The binder will be invoked automatically and each of the members will be converted.
- ***ISPF*** supports the copy of PDSE program libraries or members. The binder options are supported transparently in background (option 5.7); the foreground (option 4.7) invokes the TSO LOAD/GO Prompter which invokes the binder.
- ***SMPE*** does not support REL files for PDSEs, however copying or binding into a PDSE should be transparent to SMPE.
- ***TSO/E Test*** supports program objects in PM1 format or which have contents compatible with PM1 format. Also, it can only obtain information from those program objects for which the DCB used to load them from their program libraries is accessible. This means that TSO/E Test can not be used to test program objects that were loaded by LLA or loaded into LPA.

The following is a partial list of components and products that do not support program objects:

- Program objects are not supported by the following DFSMS utilities:
  - IEHMOVE
  - IEBDG
  - IEBGENER
  - IEBPTPCH

PDSE program library directory access of program objects

There are some changes in the way that PDSE directories can be accessed for program libraries. They include:

- PDSE program object directory entries have been extended. Information about the type of member can be obtained via the directory entry, though not as directly as ISITMGD. (Multiple tests continue to be required because the program object indicator in a program directory entry is located in the same place as the user data field for a data directory entry.)
- You can still use BLDL to access PDSE program directory entries. The format is converted to the current format, with some modifications when the program object exceeds 16 meg.
- The IHAPDS mapping, which maps the PDS directory entry information returned by the BLDL macro, has changed in order to support program objects and accommodate the >16-Meg program objects.
  - There is a bit (PDS2LFMT) which indicates that the load module is a program object and that the PDS2FTB3 flags are valid and contain additional information.
  - There is a bit (PDS2BIG) that indicates that the length field (PDS2STOR) does not hold the module length and that the large load module extension exists. The PDS2VSTR field in this extension contains the fullword load module length in this case, and PDS2STOR contains a zero.
- A second directory service, DESERV, supports both PDS and PDSE libraries. You can issue DESERV for either PDS or PDSE directory access, but you must pass the DCB address. It does not default to a predefined search order, as does BLDL. DESERV returns an SMDE that, for PDSE directories, contains more information than is mapped by IHAPDS.
- You can still read PDSE Program Library directories using BSAM. The format of each directory entry will be converted, as is done with BLDL.
- As with all PDSE, one cannot access PDSE Program Libraries using EXCP.
- Applications that need to know if a data set is a PDSE program library can issue an external macro, ISITMGD, to get this information. The data set must be open at the time. This macro is documented in *z/OS DFSMS Macro Instructions for Data Sets* and also discussed in *z/OS DFSMS: Using Data Sets*.

Migrating from the prelinker

Users presently using the prelinker-based (tactical) design can convert to the binder-based (strategic) solution with minimal effort. Recompilation of existing modules is unnecessary. Rebinding of existing support libraries, such as C370LIBs and SCEELKED, into PDSE format is unnecessary.

The two DLL designs can coexist in the same system or complex without special precautions. This will allow migration of applications to the new support, one at a time.

The binder incorporates Language Environment/370 prelinker functions

The binder incorporates the functions of the Language Environment/370 prelinker, specifically the handling of long names and support for the C WSA (writable static area) as a newly defined “deferred” class, thus removing the need for a separate prelinker step when the target program library is either a PDSE or z/OS UNIX file.

Note: The C prelinker, also known as the C pre-link utility, is currently known as the Language Environment/370 prelinker. They are all the same utility, which is referred to herein as the prelinker.

Processing with the prelinker

The output from the C or C++ compiler is an extended object file (XOBJ). As shown in Figure 42 on page 187, the prelinker then uses one or more of these XOBJ object files as input together with the prelinker control statements (INCLUDE, LIBRARY, and RENAME) to create a traditional object module. The prelinker performs autocalls for unresolved references by including object modules from PDS libraries, C370LIB libraries, or z/OS UNIX archive files.

Output from the prelinker is then fed into either the binder or linkage editor, both of which use autocall to resolve any remaining references to non-C routines. The linkage editor always creates a load module as output. The binder’s output module can be either a load module or program object, depending on whether the SYSLMOD DD statement specifies a PDS or PDSE program library or a z/OS UNIX file.

Processing without the prelinker

As before, the C/C++ compiler takes the source program and produces an XOBJ. The binder has been extended to accept not only object modules (in all structures, for example, traditional, XOBJ and GOFF), load modules, program objects and z/OS UNIX files, as earlier, but also z/OS UNIX archive files and C370LIBs for autocall functions. It also accepts all prelinker control statements. In addition, a C renaming routine was added to the existing interface validation logic in the binder. The result is that the prelinker step can be eliminated when SYSLMOD specifies a PDSE program library because all the work previously performed by the prelinker is now done by the binder. (This control flow is shown on Figure 43 on page 188.)

Eliminating the prelinker step has several advantages:

- Improved performance with the elimination of a job step
- Easier incorporation of new functions, released from the format restrictions imposed by an intermediate data structure
- Rebindable module as output, for example, it is not necessary to return to object files to rebind
- More efficient code distribution and servicing since single object files can be shipped in PTFs rather than the fully bound C module.

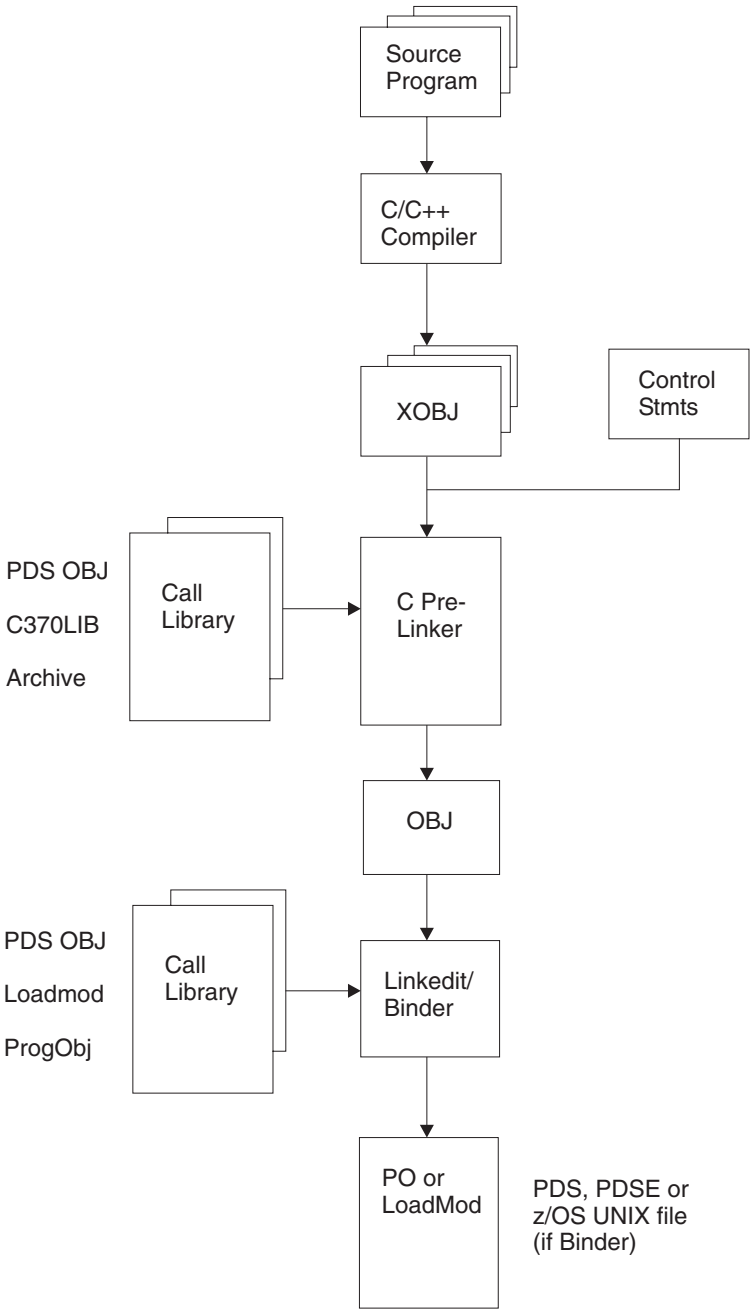


Figure 42. Invoking the prelinker. This diagram shows where the prelinker is invoked when the binder 'prelinker' function is not used.

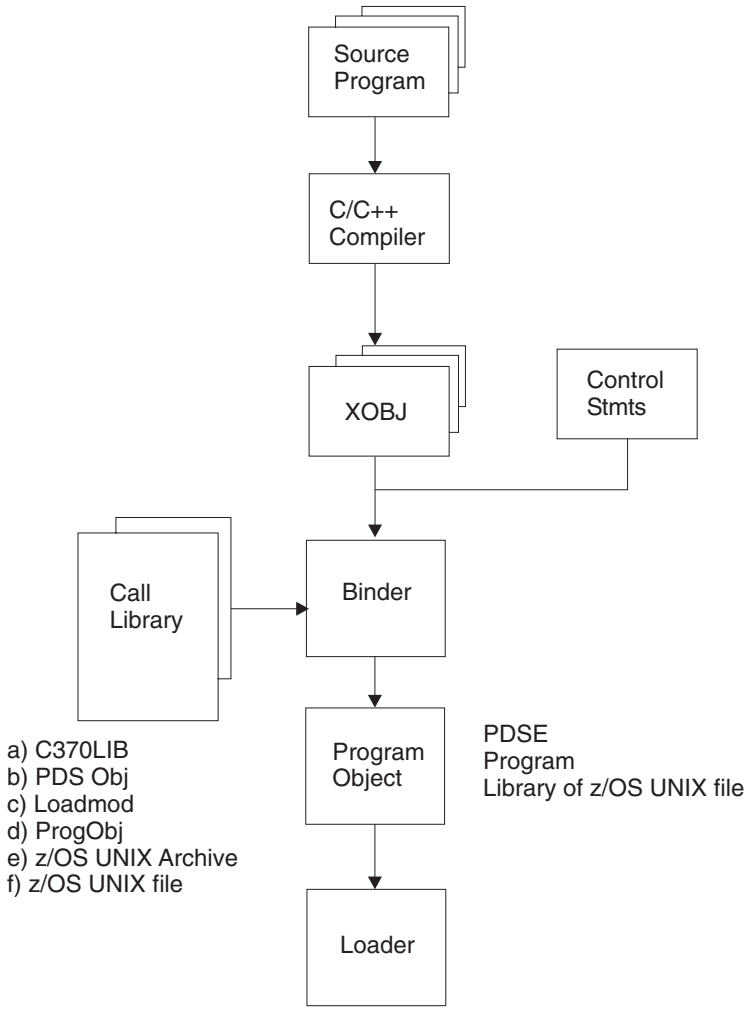


Figure 43. Prelinker elimination. This is the optional control flow in DFSMS/MVS 1.4.

Support for DLL modules in dynamic link libraries

The binder supports dynamic linking via the use of DLLs. Dynamic linking provides the ability to defer the binding of functions and variables until execution. Binder generated DLLs are program objects with a special control structure to which defines exported functions and data items. DLL-enabled applications can access (″import″) these functions and data items during execution. The creation of DLLs and DLL clients requires a language translator that can generate the requisite structures in an output XOBJ or GOFF object module.

Migrating from the prelinker and to DLLs

Migrating from the prelinker to Binder

Users must ensure that their JCL and the cataloged procedures they are invoking are changed to eliminate the prelinker step.

Note: The C89 command in z/OS UNIX will bypass the prelinker and compilers such as the IBM C/C++ compiler may provide new cataloged procedures that will use the binder for prelinker functions.

The following considerations apply if you are converting JCL yourself. It is assumed that you have already performed any necessary conversion from the linkage editor to the binder.

- The members of the SYSLIB concatenation used in the prelink step should be concatenated before the SYSLIB members used in the bind step.
- Specify CASE(MIXED) as a binder option to preserve case sensitivity.
- The contents of the prelinker SYSIN can be used as the binder SYSLIN or concatenated with it, or explicitly included by a binder INCLUDE control statement.
- SYSLMOD must be allocated to a PDSE or an z/OS file.
- If SYSDEFSD was being used for the prelinker step, it should be added to the bind step.
- If a DLL-enabled module is produced, DYNAM(DLL) must be specified. The prelinker produced a DLL-enabled module if the input XOBJ was DLL-enabled. The binder requires an explicit directive.
- If the prelinker UPCASE option was being used, it can be specified as a binder option. However it might not be necessary since the binder provides better support for long and mixed case names.
- Prelinker control statements, including RENAME and IMPORT, can be moved from the prelink step to the bind step.

Restrictions and incompatibilities migrating from the prelinker

- You must continue to use the prelinker if your target library is a PDS.
- If the prelinker is used at all, all object modules requiring prelinking must be processed together by the prelinker. In other words you cannot combine object modules created by the prelinker or load modules/program objects containing such together with XOBJ modules as input to a single bind.
- The prelinker allows names to be multiply defined, once for function names and once for variable names. The binder will use the first occurrence of a given name without regard to whether it is code or data.
- The binder does not support the version of the LIBRARY control statement that was used by the prelinker to trigger automatic library call. The unsupported

version is the one whose syntax is “LIBRARY ddname”. This is being replaced by the new binder AUTOCALL control statement.

- Code generated with the C/C++ compiler option IPA(NOLINK,NOOBJECT) should not be given as input to the binder.

Migration of applications to DLL support

Migration of applications to DLLs require that the user:

- Identify those modules that will be dynamically linked
- Recompile the DLL modules with #pragma export or the EXPORTALL option (in the C language)
- Bind those DLLs into the PDSE dynamic link library
- Remove the imported modules from the static bind library
- Rebind the application

Note: For guidance on how to create DLLs and dynamic link libraries, see *z/OS Language Environment Programming Guide*.

Appendix C. Binder return codes

The binder can be executed either as a JCL job step through TSO, through a macro call from another program, or through the binder application programming interface. The return codes are interpreted differently based on how you are executing the binder.

IEWBLINK return and reason codes

The meaning of the return codes when invoking the binder at entry point IEWBLINK are described in Table 13.

Table 13. IEWBLINK return codes

Return code	Batch execution description	Application Programming Interface (API) description
0	Informational: the program was saved and is executable.	Informational: the function was performed exactly as requested.
4	Warning: a warning condition was noted but should have no effect on the program module. Processing continues with no action required.	Warning: a warning condition was noted but should have no effect on the requested function. Processing continues with no action required.
8	Error message: The binder found an error in user data and has taken an appropriate default. The integrity of the output module is assured but might be incorrect or incomplete. The program module is saved and, if LET or LET(8) were specified, it is marked executable.	Error message: The binder found an error in user data and has taken an appropriate default. The integrity of the output parameter data is assured, but it might be null or incorrect.
12	Severe error message: the error encountered has prevented the process from completing. The resulting program module, if any, should be considered unusable.	Severe error message: the error encountered has prevented the process from completing. The function was not performed, and output parameters (except for return and reason codes) should not be used in any way.
16	Terminating error message: processing is terminated immediately.	Terminating error message: processing is terminated immediately. This return code might be accompanied by an 0F4 abend.

IEWBLDGO return codes

Table 14 contains descriptions of the return codes from the binder link-load-and-go entry point.

Table 14. IEWBLDGO return codes

Return code	Description
0	The binder linked and loaded the program, and the program executed successfully.

Table 14. IEWBLDGO return codes (continued)

Return code	Description
12	A link error occurred whose severity is greater than that specified on the LET option. The program is not loaded or executed.
16	The binder linked and loaded the program, but the program abended during execution.
n	The binder linked and loaded the program, but the program set other than a zero return code in register 15. “n” is the program’s return code.

Appendix D. Designing and specifying overlay programs

The use of overlay programs is not recommended. The information in this appendix is provided for compatibility only. Overlay programs only support load module and PM1. Therefore, any PM format later than PM1 is not supported. Program objects specifying OVLY cause the binder to create either a load module or a PM1 format program object, depending on the library type.

Ordinarily, when a program module produced by the binder is executed, all the control sections of the module remain in virtual storage throughout execution. The length of the module, therefore, is the sum of the lengths of all the control sections. When virtual storage is not at a premium, this is the most efficient way to execute a program. However, when a program approaches the limits of the available virtual storage, you could consider using the overlay facilities of the binder.

In most cases, all that is needed to convert an ordinary program to an overlay program is the addition of control statements to structure the module. You choose the portions of the program that can be overlaid, and the system arranges to load the required portions when needed during execution of the program.

When the binder overlay facility is requested, the program module is structured so that, at execution time, certain control sections are loaded only when referenced. When a reference is made from an executing control section to another, the system determines whether the code required is already in virtual storage. If it is not, the code is loaded dynamically and can overlay an unneeded part of the module already in storage.

This appendix is divided into three sections that describe the design, specification, and special considerations for overlay programs.

Note: This appendix refers to binder processing and output. These concepts also apply to linkage editor processing, unless otherwise noted, with the exception that the linkage editor cannot process program objects.

Design of an overlay program

The structure of an overlay module depends on the relationships among the control sections within the module. Two control sections do not have to be in storage at the same time to overlay each other. Such control sections are *independent*; they do not reference each other either directly or indirectly. Independent control sections can be assigned the same load addresses and are loaded only when referenced. For example, control sections that handle error conditions or unusual data can be used infrequently and need not occupy storage unless in use.

Control sections are grouped into segments. A *segment* is the smallest functional unit (one or more control sections) that can be loaded as one logical entity during execution. The control sections required all the time are grouped into a special segment called the *root segment*. This segment remains in storage throughout execution of an overlay program.

When a particular segment is executed, any segments between it and the root segment must also be in storage. This is a *path*. A reference from one segment to another segment lower in a path is a *downward reference*; the segment contains a reference to another segment farther from the root segment (see “Control section

dependency"). Conversely, a reference from one segment to another segment higher in a path (closer to the root segment) is an *upward reference*.

A downward reference might cause overlay because the necessary segment might not yet be in virtual storage. An upward reference does not cause overlay because all segments between a segment and the root segment must be present in storage.

Several paths sometimes need the same control sections. This problem can be solved by placing the control sections in another region. In an overlay structure, a *region* is a contiguous area of virtual storage within which segments can be loaded independently of paths in other regions. An overlay program can be designed in single or multiple regions.

Single region overlay program

To design an overlay structure, you should select those control sections that receive control at the beginning of execution plus those that should always remain in storage; these control sections form the root segment. The rest of the structure is developed by determining the dependencies of the remaining control sections and how they can use the same virtual storage locations at different times during execution.

The remainder of this section discusses control section dependency, segment dependency, the length of the overlay program, segment origin, communication between segments, and overlay processing.

Control section dependency

Control section dependency is determined by the requirements of a control section for <references to> or <access to> a given <routine> of <entry point> in another control section. A control section is dependent upon any control section from which it receives control or that processes its data. For example, if control section C receives control from control section B, C is dependent upon B. That is, both control sections must be in storage before execution can continue beyond a given point in the program.

Assume that a program contains seven control sections, CSA through CSG, and exceeds the amount of storage available for its execution. Before the program is rewritten, it is examined to see if it could be placed into an overlay structure. Figure 44 on page 195 shows the groups of dependent control sections in the program (the arrows indicate dependencies).

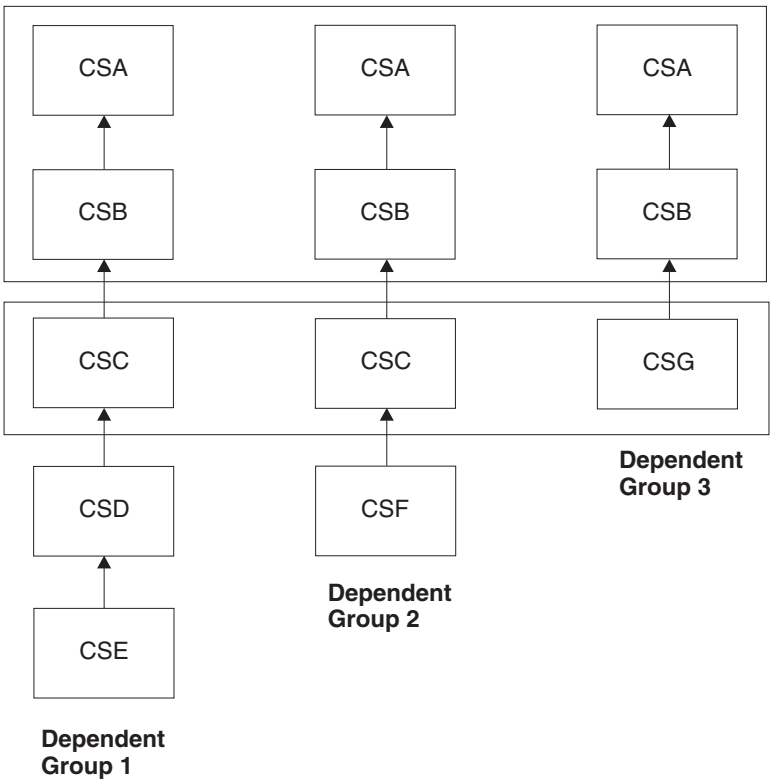


Figure 44. Control section dependencies

Each dependent group is also a path. That is, if control section CSG is executed, CSB and CSA must also be in storage. Because CSA and CSB are in each path, they must be in the root segment. Control section CSC is in two groups and therefore is a *common segment* in two different paths.

A better way to show the relationship between segments is with a tree structure. A *tree* graphically shows how segments can use virtual storage at different times. It does not imply the order of execution, although the root segment is the first to receive control. Figure 45 on page 196 shows the tree structure for the dependent groups shown in Figure 44. The structure has five segments and is contained in one region.

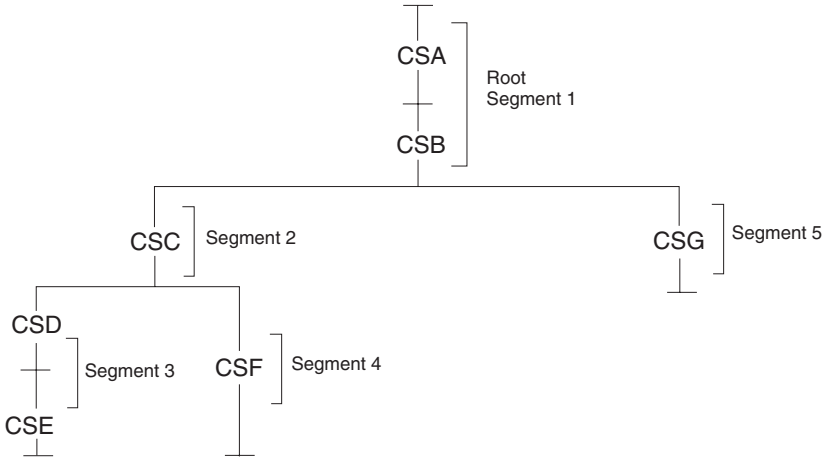


Figure 45. Single-region overlay tree structure

Segment dependency

When a segment is in virtual storage, all segments in its path are also in virtual storage. Each time a segment is loaded, all segments in its path are loaded if they are not already in virtual storage. In Figure 45, when segment 3 is in virtual storage, segments 1 and 2 are also in virtual storage. However, if segment 2 is in storage, this does not imply that segment 3 or 4 is in virtual storage because neither segment is in the path of segment 2.

The position of the segments in an overlay tree structure does not imply the sequence in which the segments are executed. A segment can be loaded and overlaid as many times as the logic of the program requires. However, a segment cannot overlay itself. If a segment is modified during execution, that modification remains only until the segment is overlaid.

Length of an overlay program

For purposes of illustration, assume the control sections in the sample program have the following lengths:

Control Section	Length (in bytes)
CSA	3000
CSB	2000
CSC	6000
CSD	4000
CSE	3000
CSF	6000
CSG	8000

If the program were not in overlay, it would require 32000 bytes of virtual storage. In overlay, however, the program requires the amount of storage needed for the longest path. In this structure, the longest path is formed by segments 1, 2, and 3, because when they are all in storage they require 18000 bytes, as shown in Figure 46.

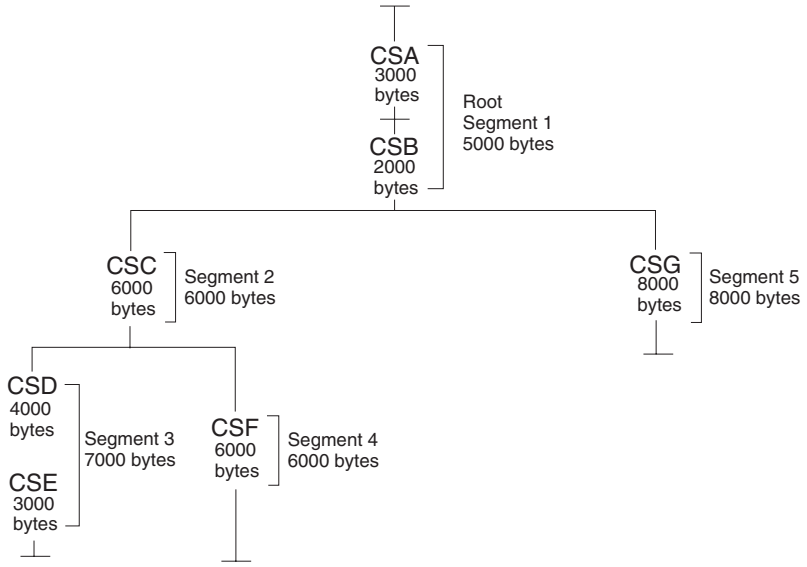


Figure 46. Length of an overlay module

Note: The length of the longest path is not the minimum requirement for an overlay program. When a program is in overlay, certain tables are used, and their storage requirements must also be considered. The storage required by these tables is described in “Special considerations” on page 209.

Segment origin

The binder assigns the relocatable origin of the root segment (the origin of the program) at 0. The relative origin of each segment is determined by 0 plus the length of all segments in the path. For example, the origin of segments 3 and 4 is equal to 0 plus 6000 (the length of segment 2) plus 5000 (the length of the root segment), or 11000. The origins of all the segments are as follows:

Segment	Origin
1	0
2	5000
3	11000
4	11000
5	5000

The segment origin is also called the *load point*, because it is the relative location where the segment is loaded.

Figure 47 on page 198 shows the segment origin for each segment and the way storage is used by the sample program. The vertical bars indicate segment origin; any two segments with the same origin can use the same storage area. This figure also shows that the longest path is that of segments 1, 2, and 3.

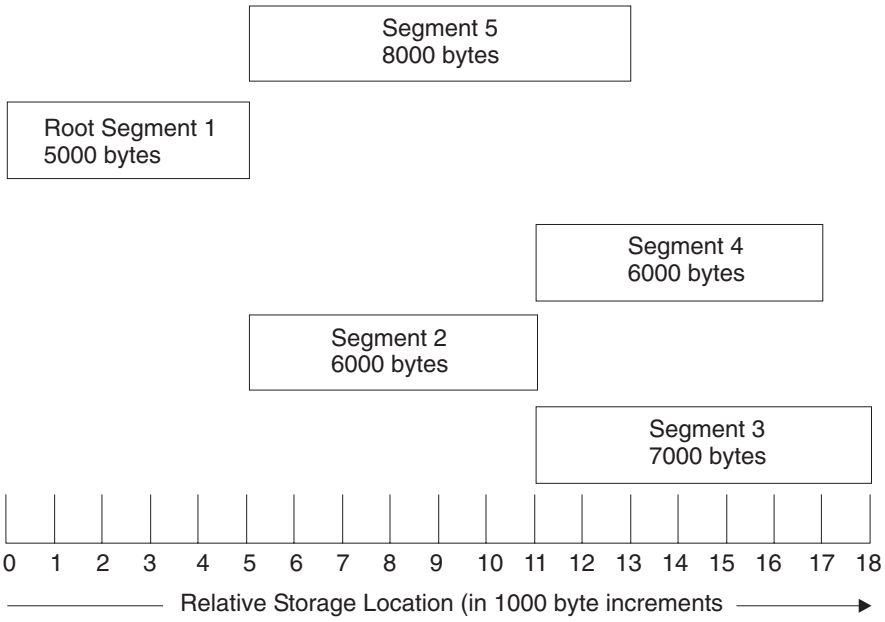


Figure 47. Segment origin and use of storage

References between segments

Segments that can be in virtual storage simultaneously are considered *inclusive*. Segments in the same region but not in the same path are considered *exclusive*; they cannot be in virtual storage simultaneously. Figure 48 shows the inclusive and exclusive segments in the sample program.

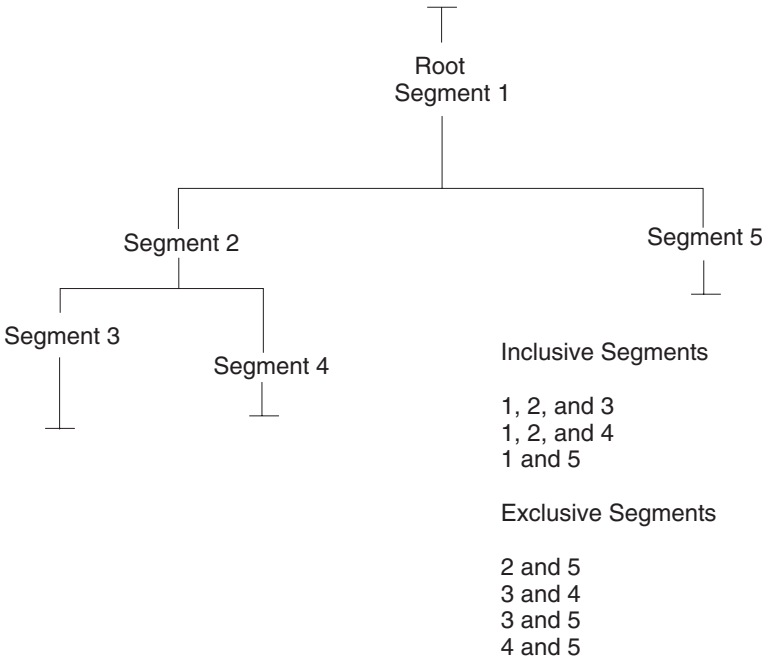


Figure 48. Inclusive and exclusive segments

Segments upon which two or more exclusive segments are dependent are called *common segments*. A segment common to two other segments is part of the path of each segment. In Figure 48 on page 198, segment 2 is common to segments 3 and 4, but not to segment 5.

An *inclusive reference* is a reference between inclusive segments, from a segment in storage to an external symbol in a segment that does not cause overlay of the calling segment. An *exclusive reference* is a reference between exclusive segments, a reference from a segment in storage to an external symbol in a segment that causes overlay of the calling segment.

Figure 49 shows the difference between an inclusive reference and an exclusive reference. The arrows indicate references between segments.

Inclusive references: Wherever possible, inclusive references should be used instead of exclusive references. Inclusive references between segments are always valid and do not require special options. When inclusive references are used, there is also less chance for error in structuring the overlay program correctly.

Exclusive references: An exclusive reference is made when the external reference in the requesting segment is to a symbol defined in a segment not in the path of the requesting segment. Exclusive references are either valid or invalid.

An exclusive reference is valid only if there is also an inclusive reference to the requested control section in a segment common to both the segment to be loaded and the segment to be overlaid. The same symbol must be used in both the common segment and the exclusive reference. In Figure 49, a reference from segment B to segment A is valid because there is an inclusive reference from the common segment to segment A. (An entry table in the common segment contains the address of segment A. The overlay does not destroy this table.)

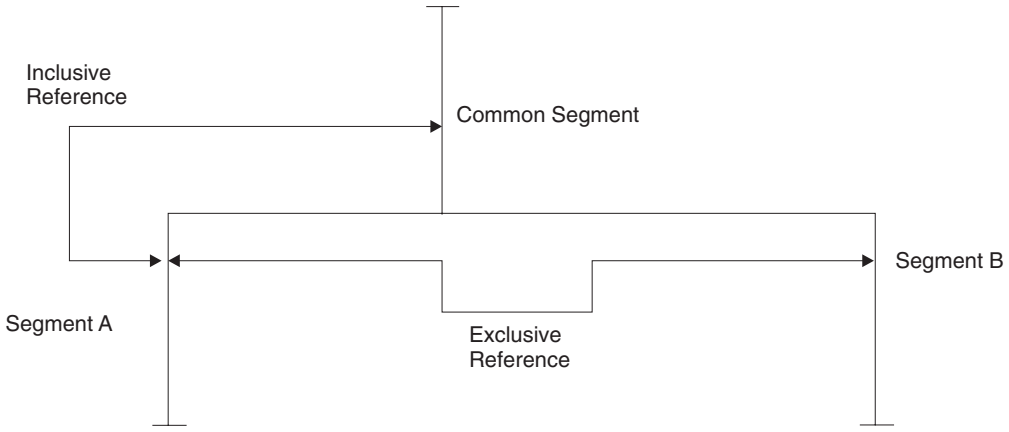


Figure 49. Inclusive and exclusive references

In this same figure, a reference from segment A to segment B is invalid because there is no reference from the common segment to segment B. A reference from segment A to segment B can be made valid by including, in the common segment, an external reference to the symbol used in the exclusive reference to segment B.

Another way to eliminate exclusive references is to arrange the program so that the references that cause overlay are made in a higher segment. For example, you could eliminate the exclusive reference shown in Figure 49 by writing a new module to be placed in the common segment. The new module's only function would be to

reference segment B. The code in segment A could then be changed to reference the new module instead of segment B. Control then would pass from segment A to the common segment, where the overlay of segment A by segment B would be initiated.

If either valid or invalid exclusive references appear in the program, the binder considers them errors unless one of the special options is used. These options are described later in this section (see “Special considerations” on page 209).

Notes:

1. During the execution of a program written in a higher level language such as Fortran, COBOL, or PL/I, an exclusive call results in abnormal termination of the program if the requested segment attempts to return control directly to the invoking segment that has been overlaid.
2. If a program written in COBOL includes a segment that contains a reference to a COBOL class test or TRANSFORM table, the segment containing the table must be in either the root segment or a segment higher in the same path than the segment containing the reference to the table.

Overlay process

The overlay process is initiated when a control section in virtual storage references a control section not in storage. The control program determines the segment that the referenced control section is in and, if necessary, loads the segment. When a segment is loaded, it overlays any segment in storage with the same relative origin. Any segments in storage that are lower in the path of the overlaid segment can also be overlaid. An exclusive reference can also cause segments higher in the path to be overlaid. No overlay occurs if a control section in storage references a control section in another segment already in storage.

The portion of the control program that determines when overlay is to occur is the *overlay supervisor* that uses special tables to determine when overlay is necessary. These tables are generated by the binder and are part of the output program module. The special tables are the segment table and the entry table(s). Figure 50 on page 201 shows the location of the segment and entry tables in the sample program.

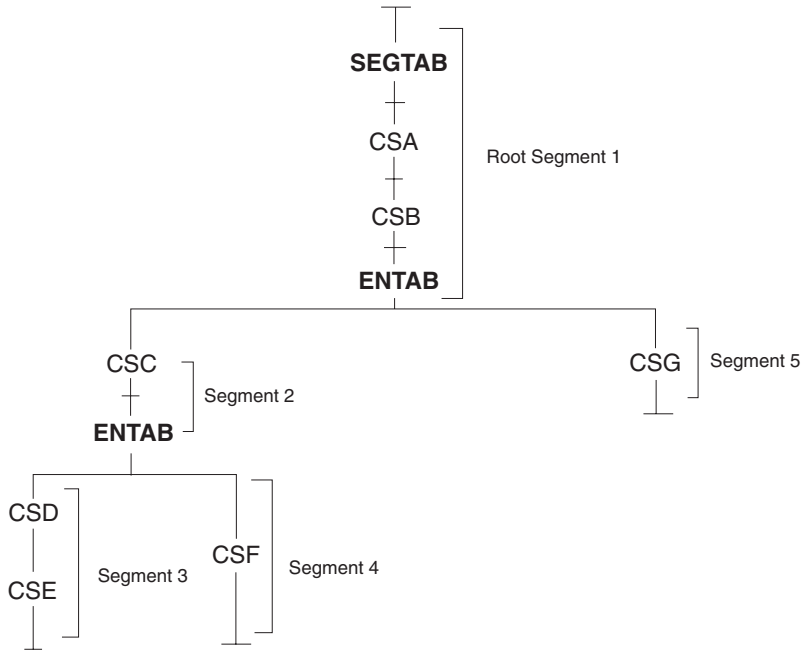


Figure 50. Location of segment and entry tables in an overlay module

Because the tables are present in every overlay module, their size must be considered when planning the use of virtual storage. The storage requirements for the tables are given in “Special considerations” on page 209. A detailed discussion of the segment and entry tables follows.

Segment table: Each overlay program contains one segment table (SEGTAB); this table is the first control section in the root segment. The segment table contains information about the relationship of the segments and regions in the program. During execution, the table also contains control information such as what segments are in storage and which are being loaded.

Entry table: Each segment that is not the last segment in a path can contain one entry table (ENTAB); when present, this table is the last control section in a segment.

When overlay is required, an entry in the table is created for a symbol to which control is passed, provided the symbol is used as an external reference in the requesting segment, and the symbol is defined in another segment either lower in the path of the requesting segment or in another region. An ENTAB entry is not created for any symbol already present in an entry table closer to the root segment (higher in the path), or for a symbol defined higher in the path. (A reference to a symbol higher in the path does not have to go through the control program because no overlay is required.)

If an external reference and the symbol it references are in segments not in the same path but in the same region, an exclusive reference was made. If the exclusive reference is valid, an ENTAB entry for the symbol is present in the common segment. Because the common segment is higher in the path of the requesting segment, no ENTAB entry is created in the requesting segment. When the reference is executed, control passes through the ENTAB entry in the common segment. That is, a branch to the location in the ENTAB entry causes the overlay supervisor to be called to load the needed segments.

If the exclusive reference is invalid, no ENTAB entry is present in the common segment. If the LET option is specified, an invalid exclusive reference causes unpredictable results when the program is executed. Because no ENTAB entry exists, control is passed directly to the relative address specified in the reference, even though the requested segment cannot be in virtual storage.

Multiple region overlay program

If a control section is used by several segments, it is usually desirable to place that control section in the root segment. However, the root segment can get so large that the benefits of overlay are lost. If some of the control sections in the root segment could overlay each other (except for the requirement that all segments in a path must be in storage at the same time), the job might be a candidate for multiple region structure. Multiple region structures can also be used to increase segment loading efficiency: processing can continue in one region while the next path to be executed is being loaded into another region.

With multiple regions, a segment has access to segments that are not in its path. Within each region, the rules for single region overlay programs apply, but the regions are independent of each other. A maximum of four regions can be used.

Figure 51 shows the relationship between the control sections in the sample program and two new control sections: CSH and CSI. The two new control sections are each used by two other control sections in different paths. Placing CSH and CSI in the root segment makes the segment larger than necessary, because CSH and CSI can overlay each other. The two control sections should not be duplicated in two paths, because the binder automatically deletes the second pair and an invalid exclusive reference might then result.

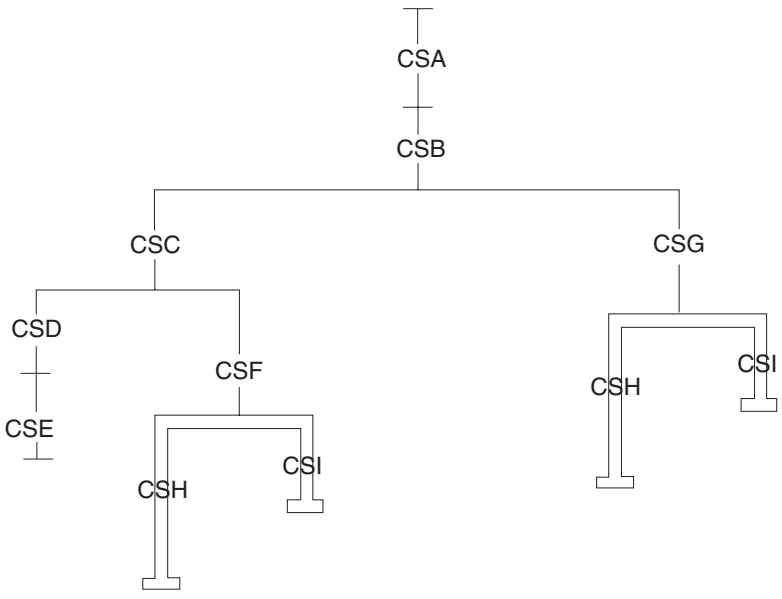


Figure 51. Control sections used by several paths

If the two control sections are placed in another region, however, they can be in virtual storage when needed, regardless of the path being executed in the first region. Figure 52 on page 203 shows all the control sections in a two-region structure. Either path in region 2 can be in virtual storage regardless of the path

being executed in region 1. Segments in region 2 can cause segments in region 1 to be loaded without being overlaid themselves.

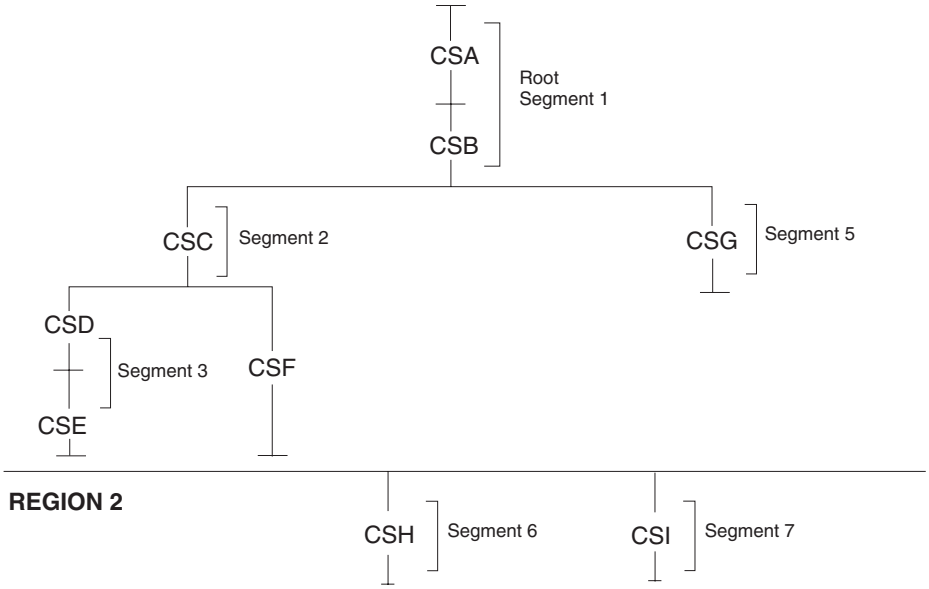


Figure 52. Overlay tree for multiple-region program

The relative origin of a second region is determined by the length of the longest path in the first region (18000 bytes). Region 2, therefore, begins at 0 plus 18000 bytes. The relative origin of a third region would be determined by the length of the longest path in the first region plus the longest path in the second region.

The virtual storage required for the program is determined by adding the lengths of the longest path in each region. In Figure 52, if CSH is 4000 bytes and CSI is 3000 bytes, the storage required is 22000 bytes, plus the storage required by the special overlay tables.

Care should be exercised when choosing multiple regions. There might be some system degradation caused by the overlay supervisor being unable to optimize segment loading when multiple regions are used.

Specification of an overlay program

Once you have designed an overlay structure, the program must be placed into that structure. You indicate to the binder the relative positions of the segments, the regions, and the control sections in each segment. Positioning is accomplished as follows:

Segments
Are positioned by OVERLAY statements. In addition, the overlay statement provides a means to equate each load point with a unique symbolic name. Each OVERLAY statement begins a new segment.

Regions
Are also positioned by OVERLAY statements. You specify the origin of the first segment of the region, followed by the word REGION in parentheses.

Control sections
Are positioned in the segment specified by the OVERLAY statement with which

they are associated in the input sequence. However, the sequence of the control sections within a segment is not necessarily the order in which the control sections are specified.

The input sequence of control statements and control sections should reflect the sequence of the segments in the overlay structure from top to bottom, left to right, and region by region. This sequence is illustrated in later examples.

In addition, several special options are used with overlay programs. These options are specified on the EXEC statement for the binder job step and are described at the end of this section.

Note: If a program module in overlay structure is reprocessed by the binder, the OVERLAY statements and special options (such as OVLY) must be specified. If the statements and options are not provided, the output program module will not be in overlay structure.

The symbolic origin of every segment, other than the root segment, must be specified with an OVERLAY statement. The first time a symbolic origin is specified, a load point is created at the end of the previous segment. That load point is logically assigned a relative address at the doubleword boundary that follows the last byte in the preceding segment. Subsequent use of the same symbolic origin indicates that the next segment is to have its origin at the same load point.

In the sample single-region program, the symbolic origin names ONE and TWO are assigned to the two necessary load points, as shown in Figure 52 on page 203. Segments 2 and 5 are at load point ONE; segments 3 and 4 are at load point TWO.

The following sequence of OVERLAY statements results in the structure in Figure 53 on page 205. (The control sections in each segment are indicated by name.)

```
Control section CSA
Control section CSB
  OVERLAY ONE
Control section CSC
  OVERLAY TWO
Control section CSD
Control section CSE
  OVERLAY TWO
Control section CSF
  OVERLAY ONE
Control section CSG
```

Note: The sequence of OVERLAY statements reflects the order of segments in the structure from top to bottom and left to right.

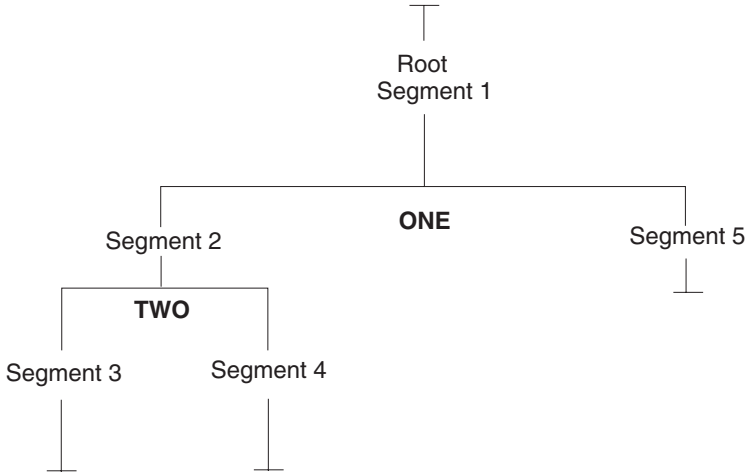


Figure 53. Symbolic segment origin in single-region program

Region origin

The symbolic origin of every region, other than the first, must be specified with an OVERLAY statement. Once a new region is specified, a segment origin from a previous region should not be specified.

In the sample multiple-region program, the symbolic origin THREE is assigned to region 2, as shown in Figure 54. Segments 6 and 7 are at load point THREE.

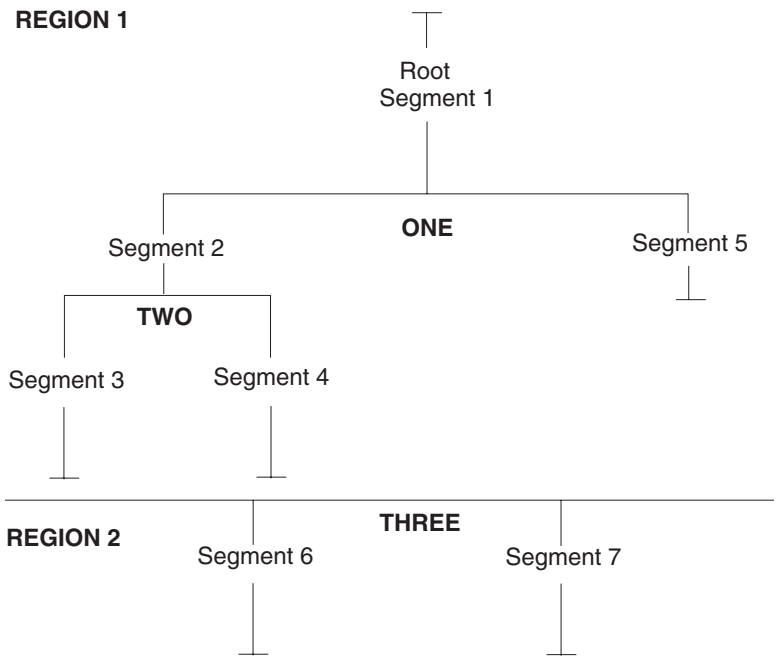


Figure 54. Symbolic segment and region origin in multiple-region program

If the following is added to the sequence for the single-region program, the multiple-region structure is produced:

```
          .
          .
          .
  OVERLAY THREE(REGION)
Control section CSH
  OVERLAY THREE
Control section CSI
```

Control section positioning

After each OVERLAY statement, the control sections for that segment must be specified. The control sections for a segment can be specified in one of three ways:

1. By placing the object decks for each segment after the appropriate OVERLAY statement
2. By using INCLUDE control statements for the modules containing the control sections for the segment
3. By using INSERT control statements to reposition a control section from its position in the input stream to a particular segment.

Any control sections that precede the first OVERLAY statement are placed in the root segment; they can be repositioned with an INSERT statement. Control sections from the automatic call library are also placed in the root segment. The INSERT statement can be used to place these control sections in another specific segment. Common areas in an overlay program are described in “Special considerations” on page 209.

An example of each of the three methods of positioning control sections follows. Each example results in the structure for the single-region sample program. An example is also given of repositioning control sections from the automatic call library.

Using object decks

The primary input data set for this example contains an ENTRY statement and seven object decks, separated by OVERLAY statements:

```
//LKED        EXEC  PGM=IEWBLINK,PARM='OVLY'
              .
              .
              .
//SYSLIN      DD    *
  ENTRY BEGIN
Object deck for CSA
Object deck for CSB
  OVERLAY ONE
Object deck for CSC
  OVERLAY TWO
Object deck for CSD
Object deck for CSE
  OVERLAY TWO
Object deck for CSF
  OVERLAY ONE
Object deck for CSG
```

The EXEC statement illustrates that the OVLY parameter must be specified for every overlay program to be processed by the binder.

Using INCLUDE statements

The primary input data set for this example contains a series of control statements. The INCLUDE statements in the primary input data set direct the binder to library members that contain the control sections of the program.

```
//LKED        EXEC  PGM=IEWBLINK,PARM='OVLY'
              .
              .
              .
//MODLIB      DD    DSNAME=USER.OBJLIB,DISP=OLD
//SYSLIN      DD    *
  ENTRY BEGIN
  INCLUDE MODLIB(CSA,CSB)
  OVERLAY ONE
  INCLUDE MODLIB(CSC)
  OVERLAY TWO
  INCLUDE MODLIB(CSD,CSE)
  OVERLAY TWO
  INCLUDE MODLIB(CSF)
  OVERLAY ONE
  INCLUDE MODLIB(CSG)
```

In this example, the control sections of the program are not part of the primary input data set, but are represented in the primary input by the INCLUDE statements. When an INCLUDE statement is processed, the appropriate control section is retrieved from the library and processed.

Using INSERT statements

When INSERT statements are used, the INSERT and OVERLAY statements can either follow or precede all the input modules. However, the order of the control sections in a segment is not necessarily the same as the order of the INSERT statements for each segment. An example of each is given, as well as an example of repositioning automatically called control sections.

Following all input: The control statements can follow all the input modules, as shown in the following example:

```
//LKED        EXEC  PGM=IEWBLINK,PARM='OVLY'
              .
              .
              .
//SYSLIN      DD    DSNAME=USER.OBJECT,DISP=OLD
//            DD    *
  ENTRY BEGIN
  INSERT CSA,CSB
  OVERLAY ONE
  INSERT CSC
  OVERLAY TWO
  INSERT CSD,CSE
  OVERLAY TWO
  INSERT CSF
  OVERLAY ONE
  INSERT CSG
```

The primary input data set contains the object modules for the control sections, and the input stream is concatenated to it.

Preceding all input: The control statements can also precede all input modules, as shown in the following example:

```
//LKED        EXEC  PGM=IEWBLINK,PARM='OVLY'
//MODULES     DD    DSNAME=USER.OBJSEQ,DISP=OLD
              .
              .
              .
//SYSLIN      DD    *
  ENTRY BEGIN
  INSERT CSA,CSB
  OVERLAY ONE
```

```
  INSERT CSC
  OVERLAY TWO
  INSERT CSD,CSE
  OVERLAY TWO
  INSERT CSF
  OVERLAY ONE
  INSERT CSG
  INCLUDE MODULES
```

The primary input data set contains all the control statements for the overlay structure and an INCLUDE statement. The data set specified by the INCLUDE statement contains all the object modules for the structure, and is a sequential data set.

Repositioning automatically called control sections: The INSERT statement can also be used to move automatically called control sections from the root segment to the desired segment. This is helpful when control sections from the automatic call library are used in only one segment. By moving such control sections, the root segment will contain only those control sections used by more than one segment.

When a program is written in a higher level language, special control sections are called from the automatic call library. Assume that the sample program is written in COBOL and that two control sections (ILBOVTR0 and ILBOSCH0) are called automatically from SYS1.COBLIB. Ordinarily, these control sections are placed in the root segment. However, INSERT statements are used in the following example to place these control sections in segments other than the root segment.

```
//LKED        EXEC  PGM=IEWBLINK,PARM='OVLY'
//MODLIB      DD    DSNAME=USER.OBJLIB,DISP=OLD
//SYSLIB      DD    DSNAME=SYS1.COBLIB,DISP=SHR
                .
                .
                .
//SYSLIN      DD    *
  ENTRY BEGIN
  INCLUDE MODLIB(CSA,CSB)
  OVERLAY ONE
  INCLUDE MODLIB(CSC)
  OVERLAY TWO
  INCLUDE MODLIB(CSD,CSE)
  INSERT ILBOVTR0
  OVERLAY TWO
  INCLUDE MODLIB(CSF)
  INSERT ILBOSCH0
  OVERLAY ONE
  INCLUDE MODLIB(CSG)
```

As a result, segments 3 and 4 contain ILBOVTR0 and ILBOSCH0 respectively.

This example also combines two ways of specifying the control sections for a segment.

Special options

The binder provides three special job step options (OVLY, LET, and XCAL) for the overlay program. These options are specified on the EXEC statement for the binder job step. They must be specified each time a program module in overlay structure is reprocessed by the binder.

OVLY option

The OVLY option must be specified for every overlay program. If the option is omitted, all the OVERLAY and INSERT statements are considered invalid, and the output module is not an overlay structure. If, in addition, the LET option is not specified, the output module is marked not executable.

LET option

The LET option allows marking the output module executable even though certain error conditions were found during binder processing. When LET is specified, any exclusive reference (valid or invalid) is accepted. At execution time, a valid exclusive reference is executed correctly; an invalid exclusive reference usually causes unpredictable results.

Also with the LET option, unresolved external references do not prevent the module from being marked executable. This could be helpful when part of a large program is ready for testing; the segments to be tested might contain references to segments not yet coded. If LET is specified, the program can be executed to test those parts that are finished (as long as the references to the absent segments are not executed). If the LET option is not specified, these unresolved references cause the module to be marked not executable.

XCAL option

With the XCAL option, a valid exclusive call is not considered an error, and the program module is marked executable. However, unless the LET option is specified, other errors could cause the module to be marked not executable. In this case, the XCAL option is not required.

AMODE and RMODE options

If the OVLY option is specified, the AMODE and RMODE options are ignored, and a diagnostic message is issued to that effect. Overlay programs are assigned as RMODE=24 and AMODE=24.

Special considerations

This section discusses several special considerations that affect overlay programs. These considerations include the handling of common areas, automatic replacement of control sections, special storage requirements, and overlay communication.

Common areas

When common areas (blank or named) are encountered in an overlay program, the common areas are collected as described previously (that is, the largest blank or identically named common area is used). The final location of the common area in the output module depends on whether INSERT statements were used to structure the program.

If INSERT statements are used to structure the overlay program, a named common area should either be part of the input stream in the segment to which it belongs or it should be placed there with an INSERT statement.

Because INSERT statements cannot be used for blank common areas, a blank common area should always be part of the input stream in the segment to which it belongs.

If INSERT statements are not used, and the control sections for each segment are placed or included between OVERLAY statements, the binder “promotes” the common area automatically. The common area is placed in the common segment of the paths that contain references to it so that the common area is in storage when needed. The position of the promoted area in relation to other control sections within the common segment is unpredictable.

If a common area is encountered in a module from the automatic call library, automatic promotion places the common area in the root segment. In the case of a named common area, this can be overridden by use of the INSERT statement.

Assume that the sample program is written in Fortran and common areas are present as shown in Figure 55. Further assume that the overlay program is structured with INCLUDE statements between the OVERLAY statements so that automatic promotion occurs.

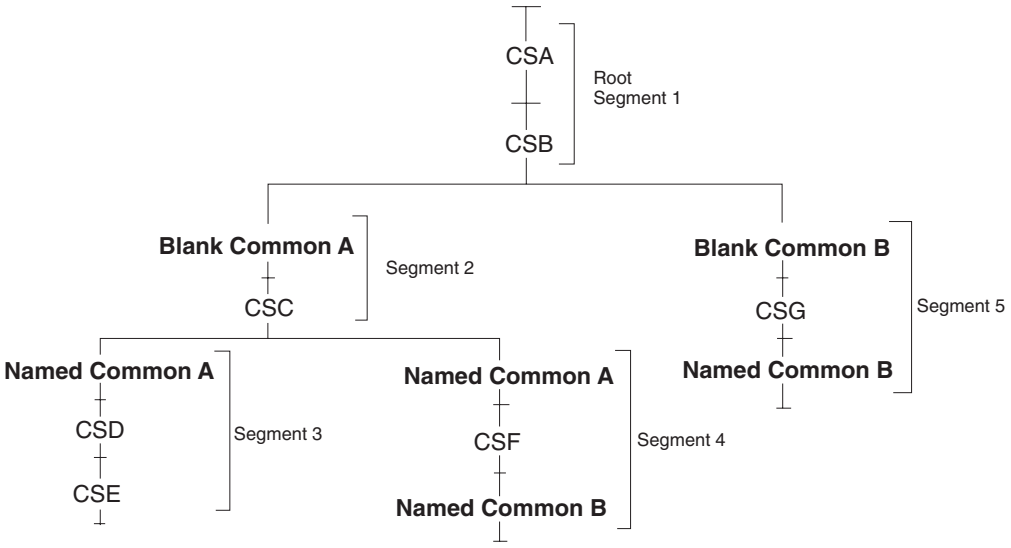


Figure 55. Common areas before processing

Segments 2 and 5 contain blank common areas. Segments 3 and 4 contain named common area A. Segments 4 and 5 contain named common area B. During binder processing, the blank common areas are collected and the larger area is promoted to the root segment (the first common segment in the two paths). The common areas named A are collected and the larger area is promoted to segment 2. The common areas named B are collected and promoted to the root segment. Figure 56 on page 211 shows the location of the common areas after processing by the binder.

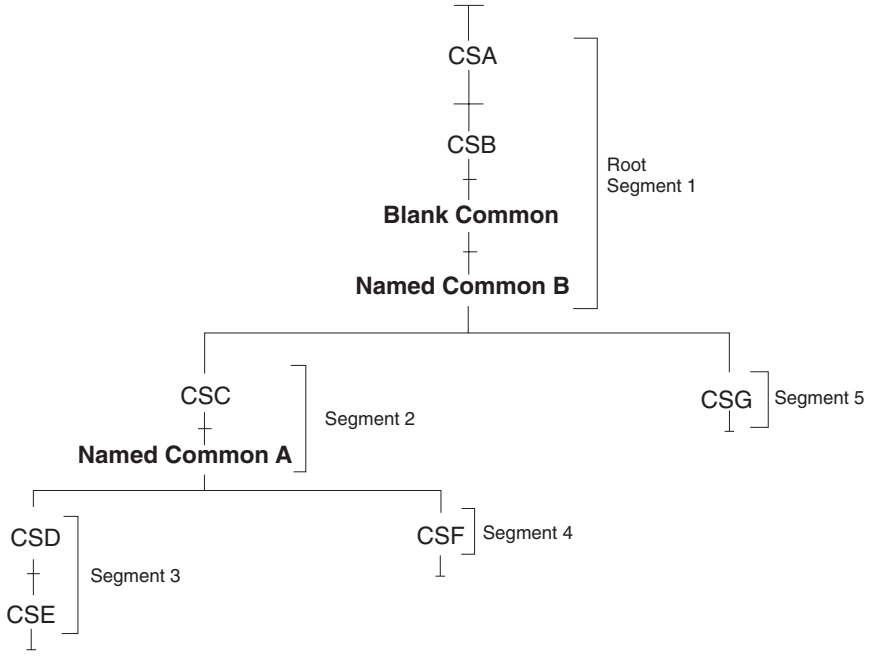


Figure 56. Common areas after processing

Automatic replacement

When identically named control sections appear in the modules of an overlay structure, the second and any subsequent control sections with that name are ignored. This occurs whether the modules are in segments in the same path or in exclusive segments. Resolution of external references might therefore cause invalid exclusive references. Invalid exclusive references cause the binder to mark the output module not executable unless the exclusive call (XCAL) option is specified on the EXEC statement (see “XCAL: Exclusive call option” on page 94).

Storage requirements

The virtual storage requirements for an overlay program include the items placed in the program by the binder.

The items that the binder places in an overlay program are the segment table, entry tables, and other control information. Their size must be included in the minimum requirements for an overlay program, along with the storage required by the longest path and any control sections from the automatic call library.

Every overlay program has one segment table in the root segment. The storage requirements are:

Length of SEGTAB = (4n + 24) bytes

Where *n* is the number of segments in the program.

Some segments will have an entry table. The requirements of the entry tables in the segments in the longest path must be added to the storage requirements for the program. The requirements for an entry table are:

Length of ENTAB = 12(x + 1) bytes

Where *x* is the number of entries in the table.

Finally, a NOTE list is required to execute an overlay program. The storage requirements are:

Length of NOTELST = (4n + 8) bytes

Where *n* is the number of segments in the program.

Overlay communication

Several ways of communicating between segments of an overlay program are discussed in this section. A higher level or assembler language program can use a CALL statement or a CALL macro instruction, respectively, to cause control to be passed to a symbol defined in another segment. The CALL can cause the segment to be loaded if it is not already present in storage. An assembler language program can also use three additional ways to communicate between segments:

1. A branch instruction that causes a segment to be loaded and control to be passed to a symbol defined in that segment.
2. A segment load (SEGLD) macro instruction, which requests loading of a segment. Processing continues in the requesting segment while the requested segment is being loaded.
3. A segment load and wait (SEGWT) macro instruction, which requests loading of a segment. Processing continues in the requesting segment only after the requested segment is loaded.

Any of the four methods can be used to make inclusive references. Only the CALL and branch can be used to make exclusive references. Do not use the SEGLD or the SEGWT macro instructions to make exclusive references. Both imply that processing is to continue in the requesting segment. An exclusive reference leads to erroneous results when the program is executed.

CALL statement or CALL macro instruction

A CALL statement or a CALL macro instruction refers to an external name in the segment where control is passed. The external name must be defined as an external reference in the requesting segment. In assembler language, the name must be defined as a 4-byte V-type address constant. The high-order bit is reserved for use by the control program and must not be altered during execution of the program.

When a CALL is used, the requested segment and any segments in its path are loaded if they are not part of the path already in virtual storage. After the segment is loaded, control is passed to the requested segment at the location specified by the external name.

A CALL between inclusive segments is always valid. A return can be made to the requesting segment by another source language statement, such as RETURN. A CALL between exclusive segments is valid if the conditions for a valid exclusive reference are met; a return from the requested segment can be made only by another exclusive reference, because the requesting segment has been overlaid.

Branch instruction

Any of the branching conventions shown in Table 15 on page 213 can be used to request loading and branching to a segment. As a result, the requested segment

and any segments in its path are loaded if they are not part of the path already in virtual storage. Control is then passed to the requested segment at the location specified by the address constant placed in general register 15.

Table 15. Branch sequences for overlay programs

Example	Name[1]	Operation	Operand[2,3]
1		L BALR	R15,=V(name) Rn,R15
2 . .		L BALR	R15,ADCON Rn,R15
	ADCON	DC	V(name)
3		L BAL	R15,=V(name) Rn,0(0,R15)[4]
4		L BAL	R15,=V(name) Rn,0(R15)[5]
5[6]		L BCR	R15,=V(name) 15,R15
6[6]		L BC	R15,=V(name) 15,0(0,R15)[4]
7[6]		L BC	R15,=V(name) 15,0(R15)[5]

Notes:

1. When the name field is blank, specification of a name is optional.
2. R15 must hold a 4-byte address constant that is the address of an entry name or a control section name in the requested segment. The address constant must be loaded into the standard entry point register, register 15.
3. Rn is any other register and is used to hold the return address. This register is usually register 14.
4. This can also be written so that the index register is loaded with the address constant; the other fields must be zero.
5. In this format, the base register must be loaded with the address constant; the displacement must be zero.
6. This example is an unconditional branch; other conditions are also allowed.

The address constant must be a 4-byte V-type address constant. The high-order byte is reserved for use by the control program and must not be altered during execution of the program. The BAS and BASR instructions cannot be used.

A branch between inclusive segments is always valid. A return can be made using the address stored in Rn. A branch between exclusive segments is valid if the conditions for a valid exclusive reference are met; a return can be made only by another exclusive reference.

Segment load (SEGLD) macro instruction

The Segment Load macro instruction provides overlap between segment loading and processing within the requesting segment. As a result of using any of the examples in Table 16, the loading of the requested segment and any segments in its path is initiated when they are not part of the path already in virtual storage. Processing then resumes at the next sequential instruction in the requesting segment while the segment or segments are being loaded. Control can be passed to the requested segment with either a CALL or a branch, as shown in Examples 1

and 2, respectively. A SEGWT instruction can be used to ensure that the data in the control section specified by the external name is in virtual storage before processing resumes, as shown in Example 3.

Table 16. Use of the SEGLD macro instruction

Example	Name[1]	Operation	Operand[2,3]
1		SEGLD CALL	external name external name
2		SEGLD branch	external name external name
3		SEGLD SEGWT L	external name external name Rn,=V(name)

Notes:

1. When the name field is blank, specification of a name is optional.
2. External name is an entry name or a control section name in the requested segment.
3. Rn is any other register and is used to hold the return address. This register is usually register 14.

The external name specified in the SEGLD macro instruction is defined with a 4-byte V-type address constant. The high-order bit is reserved for use by the control program and must not be altered during execution of the program.

Segment wait (SEGWT) macro instruction

The SEGWT macro is used to stop processing in the requesting segment until the requested segment is in virtual storage.

As a result of using any of the examples in Table 17 on page 215, no further processing takes place until the requested segment and all segments in its path are loaded when not already in virtual storage. Processing resumes at the next sequential instruction in the requesting segment after the requested segment has been loaded.

Table 17. Use of the SEGWT macro instruction

Example	Name[1]	Operation	Operand[2, 3]
1		SEGLD	external name
		SEGWT	external name
		L	Rn,ADCON
		branch	
	ADCON	DC	V(name)
2		SEGWT	external name
		L	Rn,=V(name)

Notes:

1. When the name field is blank, specification of a name is optional.
2. External name is an entry name or a control section name in the requested statement.
3. Rn is any other register and is used to hold the return address. This register is usually register 14.

If the SEGWT and SEGLD macro instructions are used together, overlap occurs between processing and segment loading. Use of the SEGWT macro instruction serves as a check to see that the necessary information is in storage when it is finally needed (see Example 1 in Table 17). In Example 2 in Table 17, no overlap is provided. The SEGWT macro instruction initiates loading, and processing is stopped in the requesting segment until the requested segment is in virtual storage.

The external name specified in the SEGWT macro instruction must be defined with a 4-byte V-type address constant. The high-order bit is reserved for use by the control program and must not be altered during execution of the program.

If the contents of a virtual storage location in the requested segment are to be processed, the entry name of the location must be referred to by an A-type address constant.

Appendix E. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Volume I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`www.ibm.com/servers/eserver/zseries/zos/bkserv/`

One exception is command syntax that is published in railroad track format; screen-readable copies of z/OS books with that syntax information are separately available in HTML zipped file form upon request to mhvrcfs@us.ibm.com.

Notices

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Programming interface information

This book primarily documents information that is NOT intended to be used as Programming Interfaces of z/OS.

This book also documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Programming Interface information

End of Programming Interface information

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, or other countries, or both:

- Advanced Function Printing
- AFP
- BookManager
- CICS OS/2
- COBOL/370
- C/370
- Database 2
- DFSMS/MVS
- DFSMSdfp
- DFSMSdss
- DFSMShsm
- DFSMSrmm
- DFSORT
- FFST
- Hiperbatch
- IBM
- IBMLink
- IMS
- Language Environment
- MVS
- MVS/DFP
- MVS/ESA
- NetSpool
- Open Class
- OS/2
- OS/390
- Parallel Sysplex
- RACF
- Resource Link
- RMF
- S/370
- SOMobjects
- System/370
- System/390
- z/OS
- zSeries

The following terms are trademarks or registered trademarks of other companies:

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary defines technical terms and abbreviations used in program management documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS manual or view *Glossary of Computing Terms*, located at:

`http://www.ibm.com/ibm/terminology`

This glossary includes terms and definitions from:

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1).

A

adata. Associated data. A collective term referring to the set of nontext, nonbinder-defined data classes stored in the program object. ADATA is used by the language and binder products to save intermediate data that can be of later use by utilities, debugging routines, etc. ADATA is not required for execution or rebinding.

A-con. A-type constant, an address

adcon. Address constant; a collective term for a field containing an address, a length, or an offset.

alias. An alternate name for a member of a partitioned data set or PDSE.

alternate entry point. A load module or program object alias for which the entry point is not the primary entry point. Other program attributes can differ within a defined alias from those of the primary entry point.

AMODE (addressing mode). The attribute of a program module that identifies whether the program entry point can receive control in 24-bit addressing mode, 31-bit addressing mode, or either.

attributes. See *program module attributes*.

automatic library call. The process by which the binder resolves external reference by including additional members from the automatic call library.

B

bind. To combine one or more control sections or program modules into a single program module, resolving references between them, or to assign virtual storage addresses to external symbols.

binder application programming interface. The set of binder entry points that allow a calling program to request specific binding and editing services individually.

binder batch interface. The set of binder entry points that allow it to perform binding and loading services.

binder dialog. A sequence of calls to the binder application programming interface to accomplish a specific task.

binder processing intent. The intended use of a binder workmod, specified at the time the workmod is created. The ACCESS processing intent indicates that the workmod will be used to copy or access program module data and that no binding will be requested. The BIND processing intent indicates that the workmod will be used to collect and edit program module data, and then bound and either saved or loaded into virtual storage for execution.

C

class. A cross section of program module data that is consistent in format and class.

Coded Character Set Identifier (CCSID). A 16-bit number that identifies a specific encoding scheme identifier, character set identifiers, code page identifiers, and additional coding required information. The CCSID uniquely identifies the coded graphic character representation used.

common area. A control section used to reserve a virtual storage area that can be referred to by other modules.

common section. Another term for *common area*.

CSECT (control section). The part of a program specified by the programmer to be an indivisible relocatable unit.

D

DFSMSdfp. A DFSMS/MVS functional component or base element of OS/390, that provides functions for storage management, data management, program management, device management, and distributed data access.

DFSMS/MVS. An IBM System/390 licensed program that provides storage, data, and device management functions. DFSMS/MVS consists of DFSMSdfp, DFSMSdss, DFSMShsm, and DFSMSrmm.

dialog. See *binder dialog*.

dialog token. A doubleword token used as an identifier for a specific binder dialog.

directory entry. A logical record in a program library directory that contains a member or alias name, a pointer to that member, and attributes of that member.

dynamic link library. A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously.

E

element. See *workmod element*.

entry point. The address or label of the first instruction executed on entering a computer program. A computer program can have a number of different entry points. The primary entry point is also called the *main* entry point.

exclusive reference. A call from a section in one overlay path to one in a different path. Because an exclusive call causes the calling section to be overlaid, return to the calling section is not possible.

exclusive segments. Segments in the same region of an overlay program that are not in the same path. Exclusive segments cannot be in virtual storage simultaneously.

external name. A name that can be referred to by any control section or separately assembled or compiled module; that is, a name that is defined in another module.

external reference. A reference to a symbol defined as an external name in another program or module.

external symbol. A control section name, entry point, common area name, part name, pseudoregister, or external reference that is defined or referred to in a particular module.

H

HFS. See *Hierarchical file system (HFS)*.

hierarchical file system (HFS). A data set that contains a POSIX-compliant file system, which is a collection of files and directories organized in a hierarchical structure, that can be accessed using z/OS UNIX System Services.

hierarchical file system (HFS) data set. A data set that contains a POSIX-compliant hierarchical file system, which is a collection of files and directories organized in a hierarchical structure, that can be accessed using z/OS UNIX System Services facilities.

I

IEWFETCH. See *program fetch*.

inclusive reference. A call from a segment in storage to an external symbol in a segment in the same path. An inclusive call does not cause overlay of the calling segment.

inclusive segments. Segments in the same region of an overlay program that are in the same path. Inclusive segments can be in virtual storage simultaneously.

intent. See *binder processing intent*.

J

J-con. An adcon containing a length.

L

load module. An executable program stored in a partitioned data set program library. See also *program object*.

loader token. An 8-byte token passed to the Program Loader to request loading of a specified deferred-load class, such as C_WSA..

M

mangled name. An external name, such as a function or variable name, which has been encoded during compilation to include type and scope information.

merge class. A deferred-load class containing only named Parts. Only the first instance of a Part is retained, but all other instances with the same name are checked to verify that they have the same length and alignment.

module map. A listing of a program module showing the length and module offset of each section.

N

name space. The set of all possible names composed of characters from the binder's character set, within which no duplicates are allowed. All external symbols have an assigned name space during binder processing and within program objects. The following name space values are defined:

- normal external names

- pseudo register names
- parts (usually external data items such as data items in C writable static).

O

object module. A collection of one or more compilation units produced by an assembler, compiler, or other language translator and used as input to the binder or linkage editor.

overlay entry table. A special section created by the binder or linkage editor at the end of an overlay segment that allows branching into an overlay segment in a different path.

overlay path. All of the segments in an overlay structure between a given segment and the root segment.

overlay program. A program module format for which some control sections occupy the same virtual storage addresses as others. The sections are organized into overlay segments, which are brought into storage as needed during execution and then overlaid by other segments when no longer needed.

overlay region. In an overlay structure, a contiguous area of virtual storage where segments can be loaded independently of paths in other regions. Only one path within a regions can be in virtual storage at any given time.

overlay segment. The smallest unit of an overlay program that can be separately loaded by the overlay supervisor. An overlay segment consists of one or more sections and is always loaded at the same offset relative to the start of the program module.

overlay segment table. A table located at the beginning of the root segment of an overlay program that describes the segments of the program.

P

page-map. A technique for loading program objects into virtual storage. The pages of a program object are brought into central storage when a page fault occurs.

part. A named subdivision of an merge class, used to describe a pseudoregister or external data item. Parts can be shared by all sections in the bound program object.

partitioned data set (PDS). A data set on direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. A PDS can contain load modules and a PDSE may contain only program objects.

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 except that a PDSE may contain only program objects or only members other than program objects.

permanent data set. A user-named data set that is normally retained for longer than the duration of a job or interactive session. Contrast with *temporary data set*.

PM1. The version of program management components that is delivered in DFSMS/MVS Version 1 Release 1 and Release 2.

PM2. The version of program management components that is delivered in DFSMS/MVS Version 1 Release 3.

PM3. The version of program management components that is delivered in DFSMS/MVS Version 1 Release 4.

PM4. The version of program management components that is delivered in z/OS Version 1 Release 3.

primary name. The name contained in the primary directory entry for a library member, used for creating, copying, and deleting the member. A library member always has one primary name and zero or more aliases.

processing intent. See *binder processing intent*.

program fetch (IEWFETCH). A program that prepares programs for execution by loading them at specific storage locations and readjusting each relocatable address constant.

program library. A partitioned data set or PDSE that always contains named members.

program management. The task of preparing programs for execution, storing the programs, load modules, or program objects in program libraries, and executing them on the operating system.

program management binder. See *binder*.

program module. The output of the binder. A collective term for *program object* and *load module*.

program module attributes. The characteristics of a program module that are stored in the program module directory entry, and are used to control the loading, rebinding, and other processing of the module.

program object. All or part of a computer program in a form suitable for loading into virtual storage for execution. Program objects are stored in PDSE program libraries and have fewer restrictions than load modules. Program objects are produced by the binder.

pseudoregister. An external dummy section used to provide global addressability to dynamically allocated control blocks, data areas, and other resources.

Q

Q-con. Q-type address constant; an offset.

R

reenterable. The reusability attribute that allows a program to be used concurrently by more than one task. A reenterable module can modify its own data or other shared resources, if appropriate serialization is in place to prevent interference between using tasks. See *reusability*.

refreshable. The reusability attribute that allows a program to be replaced (refreshed) with a new copy without affecting its operation. A refreshable module cannot be modified by itself or any other module during execution. See *reusability*.

reusability. The attribute of a module or section that indicates the extent to which it can be reused or shared by multiple tasks within the address space. See *refreshable*, *reenterable*, and *serially reusable*.

RMODE (residence mode). The attribute of a program module that identifies where in virtual storage the module is to reside (above or below 16 MB).

root segment. The first segment of an overlay program. This segment remains in virtual storage at all times during the execution of the program

S

section. A generic name given to the smallest unit of a program which can be individually manipulated during building. Sections are named by the programmer, and can be moved, replaced, or deleted during link-editing or binding.

segment. See *overlay segment*. *Class segment* is a continuous unit of text in a multiple part program object, consisting of one or more text classes, which can be separately loaded by the program loader under control of assigned loader attributes.

serially reusable. The reusability attribute that allows a program to be executed by more than one task in sequence. A serially reusable module cannot be entered by a new task until the previous task has exited. See *reusability*.

Storage Management Subsystem (SMS). A DFSMS/MVS 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.

system data. The data sets required by MVS or its subsystems for initialization and control.

system status index (SSI). A field in the directory entry of a program module which can be used to record current maintenance status.

T

temporary data set. An uncataloged data set whose name begins with & or &&, that is normally used only for the duration of a job or interactive session. Contrast with *permanent data set.*

transportable program. A program object that has been converted into a nonexecutable form for transfer to other systems.

true alias. A program alias for which the entry point is the same as the primary entry point.

U

UFS. See *UINX file system*.

UNIX file system. A section of the UNIX file tree that is physically contained on a single device or disk partition and that can be separately mounted, dismounted, and administered. Also see *hierarchical file system*.

V

V-con. V-type constant, containing an address.

W

workmod. A logical data structure in binder working storage used to assemble or otherwise operate on a program module.

workmod element. A subdivision of workmod data that is identified by a section and class name. The element is the normal unit of data transfer in binder GET and PUT data calls. See *CSECT*.

workmod token. A doubleword token used to identify a specific workmod in binder storage.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

N

O

P

R

S

T

U

V

W

X

Z

Readers' Comments — We'd Like to Hear from You

z/OS
MVS Program Management: User's Guide and Reference

Publication No. SA22-7643-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name _____ Address _____

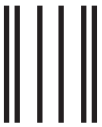
Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SA22-7643-03

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY
12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5694-A01

Printed in USA

SA22-7643-03