

English



FUJITSU Software BS2000

# BINDER V2.7A

Binder in BS2000

User Guide

Edition March 2016

## **Comments... Suggestions... Corrections...**

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:

[manuals@ts.fujitsu.com](mailto:manuals@ts.fujitsu.com)

## **Certified documentation according to DIN EN ISO 9001:2008**

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH

[www.cognitas.de](http://www.cognitas.de)

## **Copyright and Trademarks**

Copyright © 2016 Fujitsu Technology Solutions GmbH.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

---

# Contents

<b>1</b>	<b>Preface . . . . .</b>	<b>9</b>
<b>1.1</b>	<b>Objectives and target groups of this manual . . . . .</b>	<b>12</b>
<b>1.2</b>	<b>Summary of contents . . . . .</b>	<b>12</b>
<b>1.3</b>	<b>Changes since the last edition of the manual . . . . .</b>	<b>13</b>
<b>1.4</b>	<b>Notational conventions . . . . .</b>	<b>14</b>
<b>2</b>	<b>Introduction to the linker BINDER . . . . .</b>	<b>15</b>
<b>2.1</b>	<b>Link and load modules (LLMs) . . . . .</b>	<b>16</b>
<b>2.2</b>	<b>Logical structure of an LLM . . . . .</b>	<b>16</b>
<b>2.3</b>	<b>Physical structure of an LLM . . . . .</b>	<b>18</b>
2.3.1	LLMs with a single slice . . . . .	18
2.3.2	LLMs with slices by attributes . . . . .	18
2.3.3	LLMs with user-defined slices . . . . .	20
<b>2.4</b>	<b>Identification of an LLM . . . . .</b>	<b>23</b>
<b>2.5</b>	<b>Contents of an LLM . . . . .</b>	<b>29</b>
<b>2.6</b>	<b>Limiting conditions for LLMs . . . . .</b>	<b>31</b>
<b>3</b>	<b>BINDER functions . . . . .</b>	<b>35</b>
<b>3.1</b>	<b>Creating, modifying and saving an LLM . . . . .</b>	<b>37</b>
3.1.1	Creating an LLM . . . . .	37
3.1.2	Updating an LLM . . . . .	40
3.1.3	Modifying the attributes of an LLM . . . . .	43
3.1.4	Saving an LLM . . . . .	43

<b>3.2</b>	<b>Including, removing and replacing modules</b>	<b>48</b>
3.2.1	Including modules	48
3.2.2	Removing modules	52
3.2.3	Replacing modules	54
<b>3.3</b>	<b>Creating and modifying the logical structure of an LLM</b>	<b>57</b>
3.3.1	Creating the logical structure of an LLM	57
3.3.2	Modifying the logical structure of an LLM	66
<b>3.4</b>	<b>Creating the physical structure of an LLM</b>	<b>68</b>
3.4.1	User-defined slices	68
3.4.2	Slices by attributes	75
3.4.3	Modifying the type of physical structure	78
3.4.4	Connection between private and public slices	78
<b>3.5</b>	<b>Resolving external references</b>	<b>80</b>
3.5.1	Rules for resolving external references	81
3.5.2	Autolink function	85
3.5.3	Handling unresolved external references	96
<b>3.6</b>	<b>Handling name conflicts</b>	<b>102</b>
<b>3.7</b>	<b>COMMON promotion</b>	<b>103</b>
<b>3.8</b>	<b>Handling pseudo-registers</b>	<b>104</b>
<b>3.9</b>	<b>Address relocation</b>	<b>104</b>
<b>3.10</b>	<b>Handling symbols</b>	<b>105</b>
3.10.1	Symbol names	106
3.10.2	Renaming symbols	107
3.10.3	Modifying the attributes of symbols	114
3.10.4	Modifying the masking of symbols	116
3.10.5	Modifying symbol types	118
<b>3.11</b>	<b>Merging modules</b>	<b>119</b>
<b>3.12</b>	<b>Defining default values</b>	<b>125</b>
<b>3.13</b>	<b>Display functions</b>	<b>126</b>
3.13.1	Displaying the default values	126
3.13.2	Displaying symbol information	128
3.13.3	Displaying and checking library elements	131
<b>3.14</b>	<b>Controlling logging</b>	<b>133</b>
<b>3.15</b>	<b>Controlling error processing</b>	<b>134</b>
3.15.1	Severity classes	134
3.15.2	Message handling	135

---

<b>4</b>	<b>BINDER input/output</b> . . . . .	<b>137</b>
<b>4.1</b>	<b>Inputs for BINDER</b> . . . . .	<b>138</b>
<b>4.2</b>	<b>Outputs from BINDER</b> . . . . .	<b>141</b>
4.2.1	Saved LLMs . . . . .	141
4.2.2	Lists . . . . .	141
<b>5</b>	<b>BINDER run</b> . . . . .	<b>167</b>
<b>5.1</b>	<b>Calling and terminating BINDER</b> . . . . .	<b>167</b>
<b>5.2</b>	<b>Monitoring the BINDER run with job variables</b> . . . . .	<b>169</b>
<b>6</b>	<b>Subroutine interface</b> . . . . .	<b>171</b>
<b>6.1</b>	<b>BINDER macro</b> . . . . .	<b>171</b>
<b>6.2</b>	<b>Examples</b> . . . . .	<b>175</b>
<b>7</b>	<b>BINDER statements</b> . . . . .	<b>187</b>
<b>7.1</b>	<b>Grouping of statements by function</b> . . . . .	<b>187</b>
<b>7.2</b>	<b>Notes on the SDF user interface</b> . . . . .	<b>190</b>
7.2.1	SDF syntax description . . . . .	191
<b>7.3</b>	<b>Description of the statements</b> . . . . .	<b>207</b>
	BEGIN-SUB-LLM-STATEMENTS . . . . .	209
	END . . . . .	212
	END-SUB-LLM-STATEMENTS . . . . .	212
	INCLUDE-MODULES . . . . .	213
	MERGE-MODULES . . . . .	223
	MODIFY-ERROR-PROCESSING . . . . .	229
	MODIFY-LLM-ATTRIBUTES . . . . .	233
	MODIFY-MAP-DEFAULTS . . . . .	239
	MODIFY-MODULE-ATTRIBUTES . . . . .	242
	MODIFY-STD-DEFAULTS . . . . .	247
	MODIFY-SYMBOL-ATTRIBUTES . . . . .	253
	MODIFY-SYMBOL-TYPE . . . . .	257
	MODIFY-SYMBOL-VISIBILITY . . . . .	259
	REMOVE-MODULES . . . . .	262

---

RENAME-SYMBOLS . . . . .	263
REPLACE-MODULES . . . . .	268
RESOLVE-BY-AUTOLINK . . . . .	277
SAVE-LLM . . . . .	285
SET-EXTERN-RESOLUTION . . . . .	299
SET-USER-SLICE-POSITION . . . . .	301
SHOW-DEFAULTS . . . . .	303
SHOW-LIBRARY-ELEMENTS . . . . .	305
SHOW-MAP . . . . .	312
SHOW-SYMBOL-INFORMATION . . . . .	321
START-LLM-CREATION . . . . .	323
START-LLM-UPDATE . . . . .	329
START-STATEMENT-RECORDING . . . . .	334
STOP-STATEMENT-RECORDING . . . . .	335
<b>8 Usage models for generating LLMs . . . . .</b>	<b>337</b>
<hr/>	
<b>8.1 Program . . . . .</b>	<b>338</b>
<b>8.2 Module . . . . .</b>	<b>340</b>
<b>8.3 Program library . . . . .</b>	<b>342</b>
<b>8.4 Module library . . . . .</b>	<b>344</b>
<b>8.5 Generating the different program types . . . . .</b>	<b>346</b>
8.5.1 Generating a program . . . . .	346
8.5.1.1 Non-shareable program . . . . .	346
8.5.1.2 Partially shareable program . . . . .	347
8.5.1.3 Totally shareable program . . . . .	349
8.5.2 Generating a module . . . . .	351
8.5.2.1 Non-shareable module . . . . .	351
8.5.2.2 Partially shareable module . . . . .	353
8.5.2.3 Totally shareable module . . . . .	355
8.5.3 Generating a program library . . . . .	357
8.5.4 Generating a module library . . . . .	359
<b>8.6 More information about the usage models . . . . .</b>	<b>360</b>
8.6.1 LLM format 1 . . . . .	360
8.6.2 Preloading public slices via the ASHARE interface . . . . .	360

<b>9</b>	<b>Migration . . . . .</b>	<b>363</b>
<b>9.1</b>	<b>Old and current concepts . . . . .</b>	<b>363</b>
9.1.1	Old concepts . . . . .	363
9.1.2	Current concepts . . . . .	365
<b>9.2</b>	<b>Features of the current Binder-Loader-Starter system . . . . .</b>	<b>366</b>
9.2.1	Link and load module (LLM) and BINDER . . . . .	366
9.2.2	Dynamic binder loader DBL . . . . .	367
9.2.3	DBL and BINDER . . . . .	368
<b>10</b>	<b>Appendix: Description of the ISAM keys . . . . .</b>	<b>369</b>
	<b>Glossary . . . . .</b>	<b>373</b>
	<b>Related publications . . . . .</b>	<b>383</b>
	<b>Index . . . . .</b>	<b>385</b>





---

# 1 Preface

A source program processed by a compiler (Assembler, C, COBOL, FORTRAN, PL1 etc.) may exist in either object module format or link and load module format. Object modules (OMs) and link and load modules (LLMs) are the input objects for the **Binder-Loader-Starter** system, which generates executable programs from these objects.

The **binder** links the translated source program to other object modules or link and load modules to produce a loadable unit. To do this it locates the object modules and link and load modules required for the program run and links them. It also resolves cross-references between the modules, i.e. adjusts the addresses which reference fields in other modules (external references) and therefore could not be entered by the language processor at compilation or assembly time. This procedure is known as link editing.

A **loader** is needed to bring the unit generated by link editing into computer memory. Only then can the program be run.

BINDER belongs to the **Binder-Loader-Starter (BLS)** system, which also provides the user with the following functional units:

- the subsystem BLSSERV with the functionality of the dynamic binder loader DBL
- the security component BLSSEC (which may be optionally activated)

## The linker BINDER

BINDER is a linkage editor which links modules into a loadable unit with a logical and physical structure. This unit is referred to as a **link and load module (LLM)**. BINDER stores the LLM as a type L library element in a program library or in a PAM file.

Modules linked by BINDER into an LLM may be:

- object modules (OMs) and prelinked OMs from an object module library (OML), from a program library (type R) or from the temporary EAM object module file (OMF)
- prelinked LLMs, or LLMs generated by compilers, from a program library (type L)
- prelinked LLMs from a PAM file (PAM-LLM)

## BLSSERV with the dynamic binder loader DBL

The **dynamic binder loader (DBL)** links modules into a load unit and loads this into memory. The DBL functionality is part of the BLSSERV subsystem.

Modules linked by DBL into a load unit may be:

- link and load modules (LLMs) linked by BINDER or generated by compilers and stored in a program library (type L)
- link and load modules (LLMs) linked by BINDER and stored in a PAM file (PAM-LLMs, as of BLSSERV V2.5)
- object modules (OMs) generated by compilers and stored in an object module library (OML), in a program library (type R) or in the temporary EAM object module file

## The security component BLSSEC

If a “secure system” is required, the security component BLSSEC can be optionally loaded as a subsystem. This causes the Binder Loader Starter system to run a security check before each object is loaded by DBL and thus ensures that the object is loaded only if no problems have occurred. Activating the BLSSEC subsystem does, however, reduce the loading efficiency for all load calls to BLS, so this subsystem should normally be unloaded after a successful security check.

The following table shows which modules are processed or created by the individual functional units. Figure 1 shows the interaction between these functional units.

Type of module	System module		
	BINDER	DBL	BLSSEC
Object module (OM)	yes	yes	yes
Link and load module (LLM)	yes	yes	yes
Link and load module in PAM file (PAM-LLM)	yes	yes	yes
Prelinked module	yes	yes	yes
Program (load module)	no	no	yes

The linkage editor BINDER is a utility routine. The dynamic binder loader DBL, in contrast, belongs to the subsystem BLSSERV which is a component of the BS2000 Control System. It offers its functions via BS2000 commands and via program interfaces. Execution of a loaded program is initiated by a starter program which is a component of the BLSSERV subsystem and is not visible to the user.

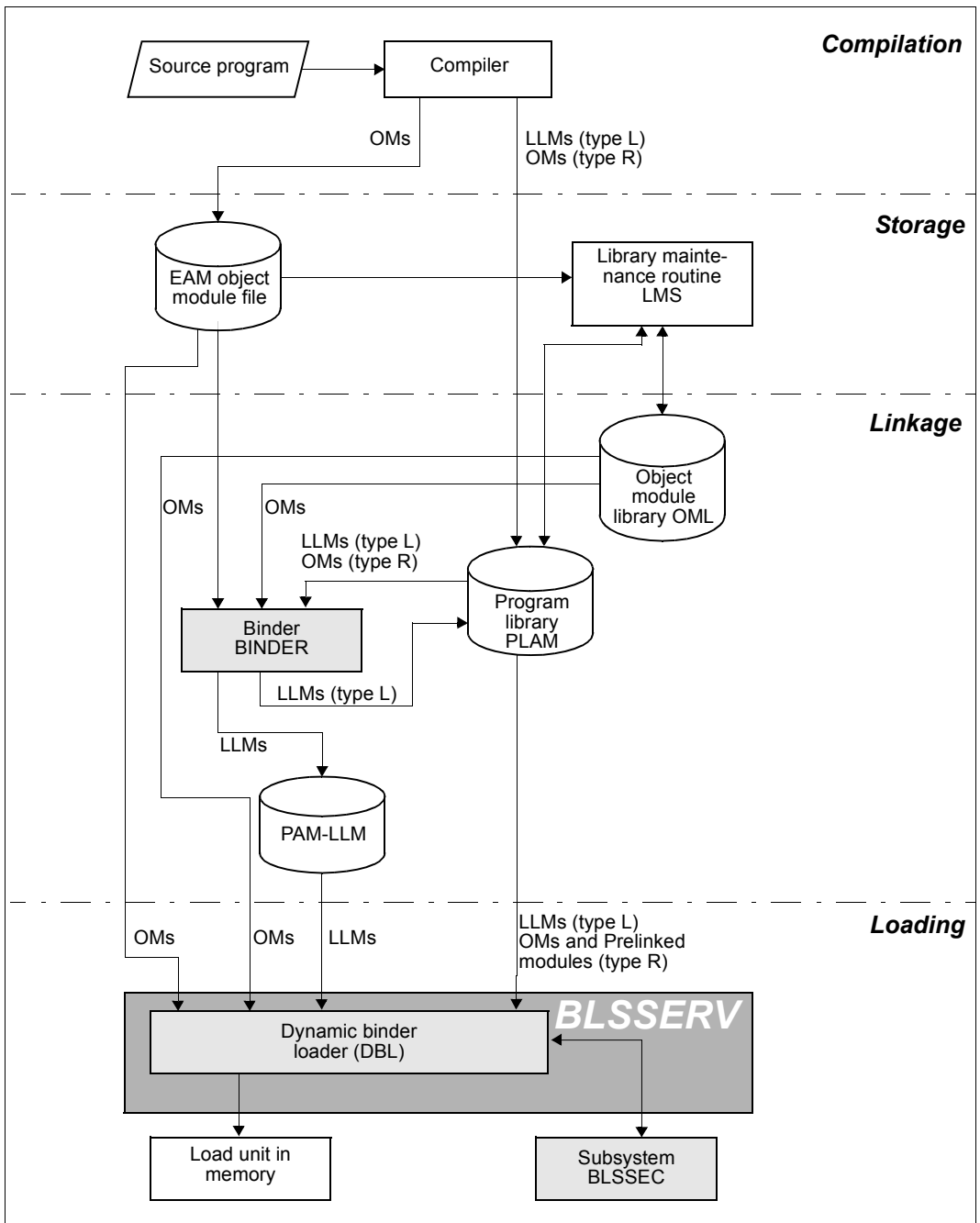


Figure 1: Interaction of the functional units for linking and loading

## 1.1 Objectives and target groups of this manual

This manual for the linker BINDER is addressed to software developers. The manual provides a description of the facilities and use of BINDER, and is also intended to serve as a reference work for all BINDER statements and the macro.

A separate Ready Reference is also available for BINDER. This contains the formats of all BINDER statements and of the BINDER macro. The Ready Reference is intended as a quick overview for users who are already familiar with BINDER.

## 1.2 Summary of contents

The description of the entire Binder-Loader-Starter (BLS) system is divided into three manuals:

- This manual describes the linker BINDER with its functions, statements and subroutine interface.
- The “BLSSERV Dynamic Binder Loader / Starter” manual [1] contains the description of the dynamic binder loader DBL.

This manual is organized as follows:

- The first four chapters describe the structures and contents of link and load modules (LLMs), together with the functions and input/output of BINDER.
- The next three chapters deal with the BINDER run, the subroutine interface and the BINDER statements in the form of a reference section.
- The following chapter presents various usage models for LLMs and explains how LLMs must be generated to comply with these models.
- The chapter “Migration” lists the main differences between the old linkage editor/loader concept (used up to BS2000 V9.5) and the new Binder-Loader-Starter system introduced with BS2000 V10.0 and is intended to help the user make the transition.
- All the BINDER messages, with explanations of their meanings and possible remedial action to be taken by the user, and a Glossary of important BINDER terms are provided at the end of the manual.



For information on the linkage editor TSOSLNK and the static loader ELDE as well as information on migration from TSOSLNK to BINDER refer to the previous version of the manual (“BINDER V2.3”).

## Readme file

The functional changes to the current product version and revisions to this manual are described in the product-specific Readme file.

Readme files are available to you online in addition to the product manuals under the various products at <http://manuals.ts.fujitsu.com>. You will also find the Readme files on the Softbook DVD.

### *Information under BS2000*

When a Readme file exists for a product version, you will find the following file on the BS2000 system:

```
SYSRME.<product>.<version>.<lang>
```

This file contains brief information on the Readme file in English or German (<lang>=E/D). You can view this information on screen using the `/SHOW-FILE` command or an editor. The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the user ID under which the product's files are stored.

### *Additional product information*

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at <http://manuals.ts.fujitsu.com>.

## 1.3 Changes since the last edition of the manual

The current version of the "BINDER" manual incorporates the following changes compared to the previous version ("BINDER V2.3"):

- For PAM-LLM, in the SAVE-LLM statement: New operand \*BEHIND-SLICE(SLICE = <structured-name 1..32>) in the structure of the LOAD-ADDRESS=\*BY-SLICES operand for specifying the SLICE-ADDRESS.
- START-LLM-CREATION and MODIFY-LLM-ATTRIBUTES statements: New RELOAD-SLICE operand in the structure of the BY-USER(AUTOMATIC-CONTROL) operand for reloading a slice that is already loaded.
- Support of X86 processors: The values X86 and MIXX8 for HSI are displayed in the map.
- MODIFY-MAP-DEFAULTS and SHOW-MAP statements: New value \*X86 for the HSI-CODE operand for supporting X86 in the GENERATE-COPYRIGHT-MODULE

- SHOW-LIBRARY-ELEMENT statement: If the SELECT operand is set to \*NAME-COLLISION, BINDER searches for duplicates also in the LLM currently in process.
- RENAME-SYMBOLS statement: New WARNING-MESSAGE operand in the structure of the SYMBOL-OCCURRENCE operand for improving the diagnose ability.
- BINDER output: New list example for modules included from a SYSPLAMALT variable

### General information

The name BS2000/OSD-BC for the BS2000 basic configuration has changed and from Version V10.0 becomes: BS2000 OSD/BC.

## 1.4 Notational conventions

The following notational conventions are used in this manual:

- References in the text to other publications are given in an abbreviated form. The full titles of all publications referred to can be found under “Related publications” at the back of the manual. This includes instructions for ordering these publications.
- In the examples, user input appears in **Courier** typeface and system output in ordinary Courier typeface.



This symbol denotes important information which you should always observe.

---

## 2 Introduction to the linker BINDER

BINDER is a new linkage editor which links together modules to produce **link and load modules (LLMs)** and stores these as library elements (element type L) in a program library.

A link and load module (LLM) is an object with the characteristics of prelinked modules, providing dynamic linking/loading.

Like a prelinked module, an LLM comprises a number of modules that are linked by BINDER. An LLM similarly incorporates the full functionality of load modules, e.g. overlays and core image format.

The main features of the BINDER concept are summarized below:

1. BINDER processes LLMs not only in batch mode but also in interactive mode. This means:
  - Each statement is processed *immediately* as it is entered.
  - When creating an LLM, the user can request information about the current status of the LLM at any time. The user can then decide whether to include further modules in the current LLM or to remove modules.
2. LLMs that are stored in program libraries can be reused either in their entirety or partially. This means:
  - A complete LLM or a “sub-LLM” can be included in the current LLM.
  - An LLM that is stored in a program library can be updated; modules can be included in the LLM or reorganized within the LLM during this process.
3. During a single BINDER run, *more than one* LLM can be created or updated, i.e. the BINDER run is not terminated after the first LLM has been saved. In addition, the same LLM can be saved multiple times, e.g. with different characteristics in the same program library or in different program libraries.

The structure and characteristics of a link and load module (LLM) are described in detail in the following section.

## 2.1 Link and load modules (LLMs)

A link and load module (LLM) comprises one or more **modules**. It is saved by BINDER as a library element of **element type L** in a program library.

An LLM may incorporate the following modules:

- object modules (OMs) created by compilers
- existing link and load modules (LLMs) in a program library (element type L).

Prelinked modules have the same format as OMs created by compilers. They are therefore regarded as OMs in the following.

An LLM has the following attributes:

- a logical structure
- a physical structure
- an identification
- contents.

## 2.2 Logical structure of an LLM

The logical structure of an LLM is implemented in the form of a tree. This tree structure comprises the following elements (see [figure 2](#)):

1. The root, which is represented by the **internal name** (INTERNAL-NAME) of the LLM. The internal name is referenced in the statement and operand descriptions.
2. The nodes, which are formed by substructures known as **sub-LLMs**. The sub-LLMs are organized hierarchically in levels.
3. The leaves, which are formed by object modules (OMs) and empty sub-LLMs that are linked in to the LLM.

Advantages of the logical structure are:

- Sub-LLMs of an LLM can be individually included, removed or replaced since each sub-LLM can be addressed directly (see [page 48ff](#)).
- When resolving external references in an LLM, the scope of action can be restricted, e.g. to the sub-LLM on the lowest level, since each sub-LLM can be searched as a separate unit.



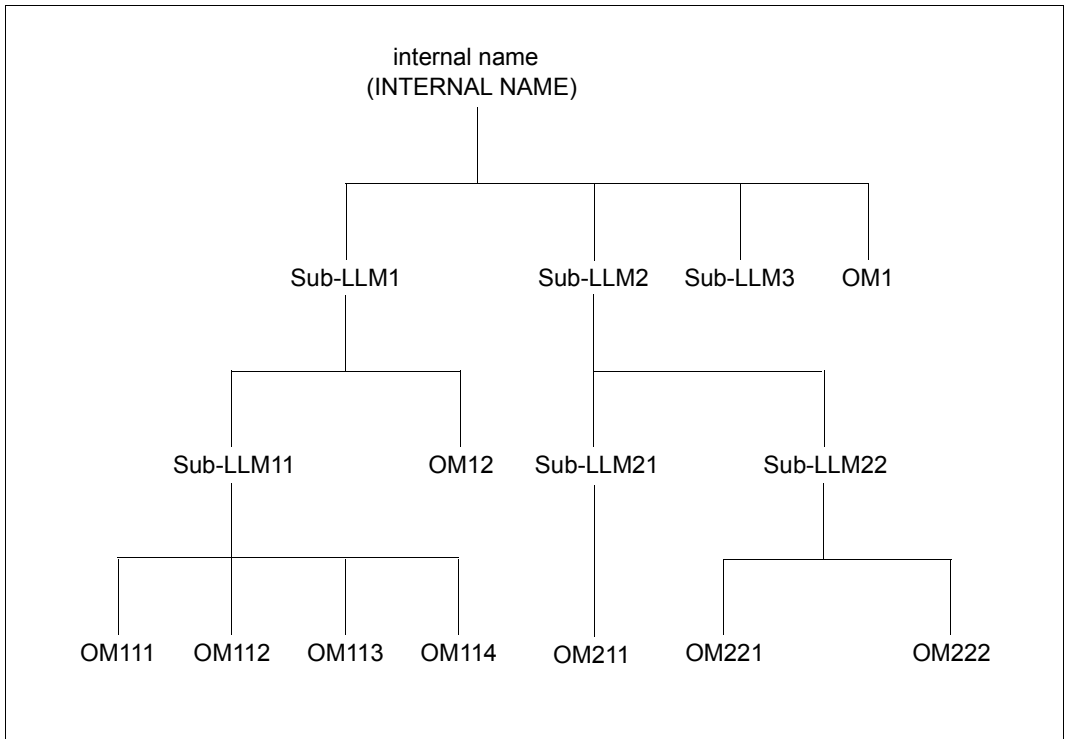


Figure 2: Example of the logical structure of an LLM

## 2.3 Physical structure of an LLM

The object modules (OMs) of an LLM consist of control sections (CSECTs) that the dynamic binder loader (DBL) loads into main memory as independent entities. With an LLM, however, the user has the opportunity to combine the CSECTs from one or more OMs of the LLM to form a single unit that allows all the CSECTs to be loaded contiguously into main memory. A loadable unit of this type is known as a **slice**. The slices form the physical structure of an LLM. There are no limitations on the size of a slice. We differentiate between the following three types of physical structure for LLMs:

- LLMs with a single slice
- LLMs with slices formed on the basis of attributes of CSECTs (slices by attributes)
- LLMs with slices defined by the user (user-defined slices).

The user specifies the type of physical structure when creating an LLM with the START-LLM-CREATION statement. The physical structure can be modified with the aid of the MODIFY-LLM-ATTRIBUTES statement.

### 2.3.1 LLMs with a single slice

The LLM consists of a single slice. There are no overlays.

### 2.3.2 LLMs with slices by attributes

Certain **attributes** can be applied to the data and instructions of CSECTs; these are evaluated during linking and loading.

If requested by the user, BINDER will combine all CSECTs having the same attributes or the same combination of attributes to form a slice. A single CSECT cannot be split up over more than one slice. It is always contained in one slice.

BINDER forms slices on the basis of the following attributes:

#### *READ-ONLY*

- Read access (READ-ONLY=YES)  
The CSECT can only be read. This attribute protects the CSECT in main memory against overwriting.
- Read and write access (READ/WRITE) (READ-ONLY=NO)  
The CSECT can be read and overwritten.

#### *RESIDENT*

- Main memory resident (RESIDENT=YES)  
The CSECT is loaded into class 3 memory and held resident there.
- Pageable (PAGEABLE) (RESIDENT=NO)  
The CSECT is pageable.  
  
This attribute is relevant *only* for system shared code.

#### *PUBLIC*

- Shareable (PUBLIC=YES)  
The CSECT contains data and instructions available for shared use. A slice created from CSECTs with the attribute PUBLIC may be loaded in a Common Memory Pool (see the “BLSSERV Dynamic Binder Loader / Starter” manual [1]) or as an unprivileged subsystem (see the “Introductory Guide to Systems Support” [9]).
- Nonshareable (PRIVATE) (PUBLIC=NO)  
The CSECT contains data and instructions available for private use only.

#### *RMODE*

- Residence mode (RMODE=ANY)  
The CSECT can be loaded below 16 Mb and above 16 Mb.
- Residence mode (RMODE=24)  
The CSECT can be loaded below 16 Mb only.

These attributes can be combined as desired. A maximum of 16 slices is permitted. The names of the slices are determined by BINDER (see [page 77](#)).

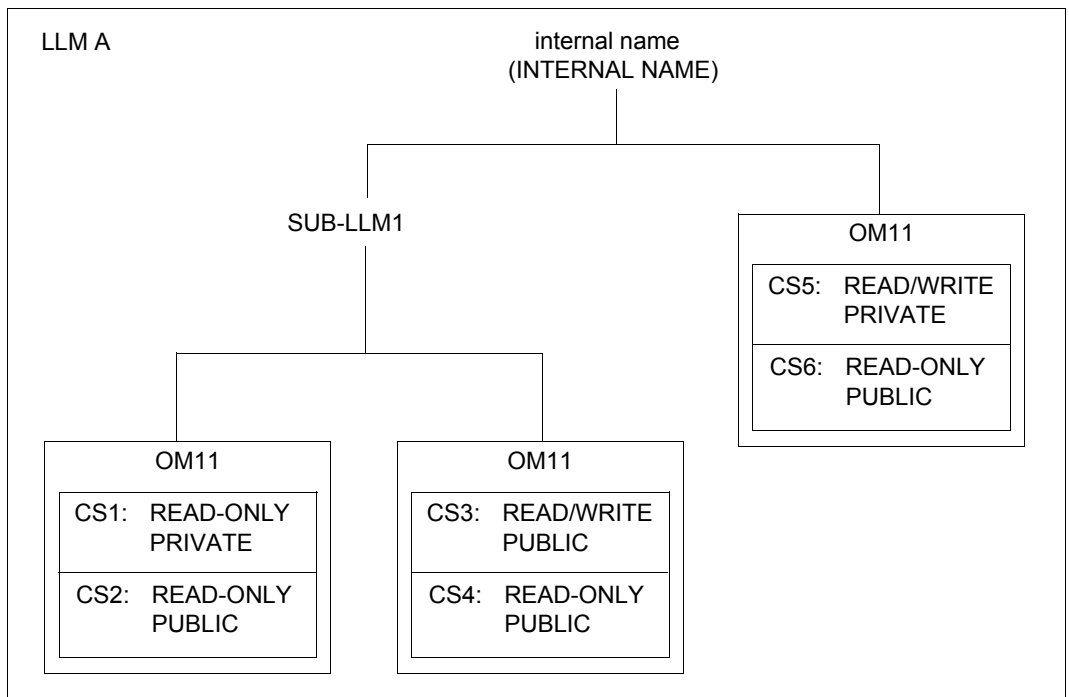
### 2.3.3 LLMs with user-defined slices

The physical structure of the LLM is determined by the user by means of SET-USER-SLICE-POSITION statements. Overlay structures can also be defined (see [page 68ff](#)). The CSECTs of a single object module (OM) cannot be split up over more than one slice. They are always contained in one slice.

#### Example

Let us assume there is an LLM A with the following *logical* structure:

OM	CSECT	CSECT attributs
OM11	CS1 CS2	READ-ONLY and PRIVATE READ-ONLY and PUBLIC
OM12	CS3 CS4	READ/WRITE and PUBLIC READ-ONLY and PUBLIC
OM1	CS5 CS6	READ/WRITE and PRIVATE READ-ONLY and PUBLIC



The following *physical* structures of LLM A can be defined, for example (see [figure 3](#)).

*Single slice*

The following are combined to form the single slice SLICE1:

- CSECTs CS1 and CS2 from OM11,
- CSECTs CS3 and CS4 from OM12 and
- CSECTs CS5 and CS6 from OM1.

The READ-ONLY and PUBLIC attributes of the CSECTs are ignored.

*Slices by attributes*

The following slices are formed:

- SLICE1 from all CSECTs having the attributes READ-ONLY and PUBLIC. These are CSECTs CS2 from OM11, CS4 from OM12 and CS6 from OM1.
- SLICE2 from all CSECTs having the attributes READ/WRITE and PUBLIC. This is CSECT CS3 from OM12.
- SLICE3 from all CSECTs having the attributes READ-ONLY and PRIVATE. This is CSECT CS1 from OM11.
- SLICE4 from all CSECTs having the attributes READ/WRITE and PRIVATE. This is CSECT CS5 from OM1.

*User-defined slices*

The following overlay structure is specified:

- CSECTs CS1 and CS2 from OM11 are combined to form SLICE1. SLICE1 is to be the root slice in the overlay structure.
- CSECTs CS3 and CS4 from OM12 are combined to form SLICE2. SLICE2 is to be contiguous with SLICE1.
- CSECTs CS5 and CS6 are combined to form SLICE3. SLICE3 is to overlay SLICE2.

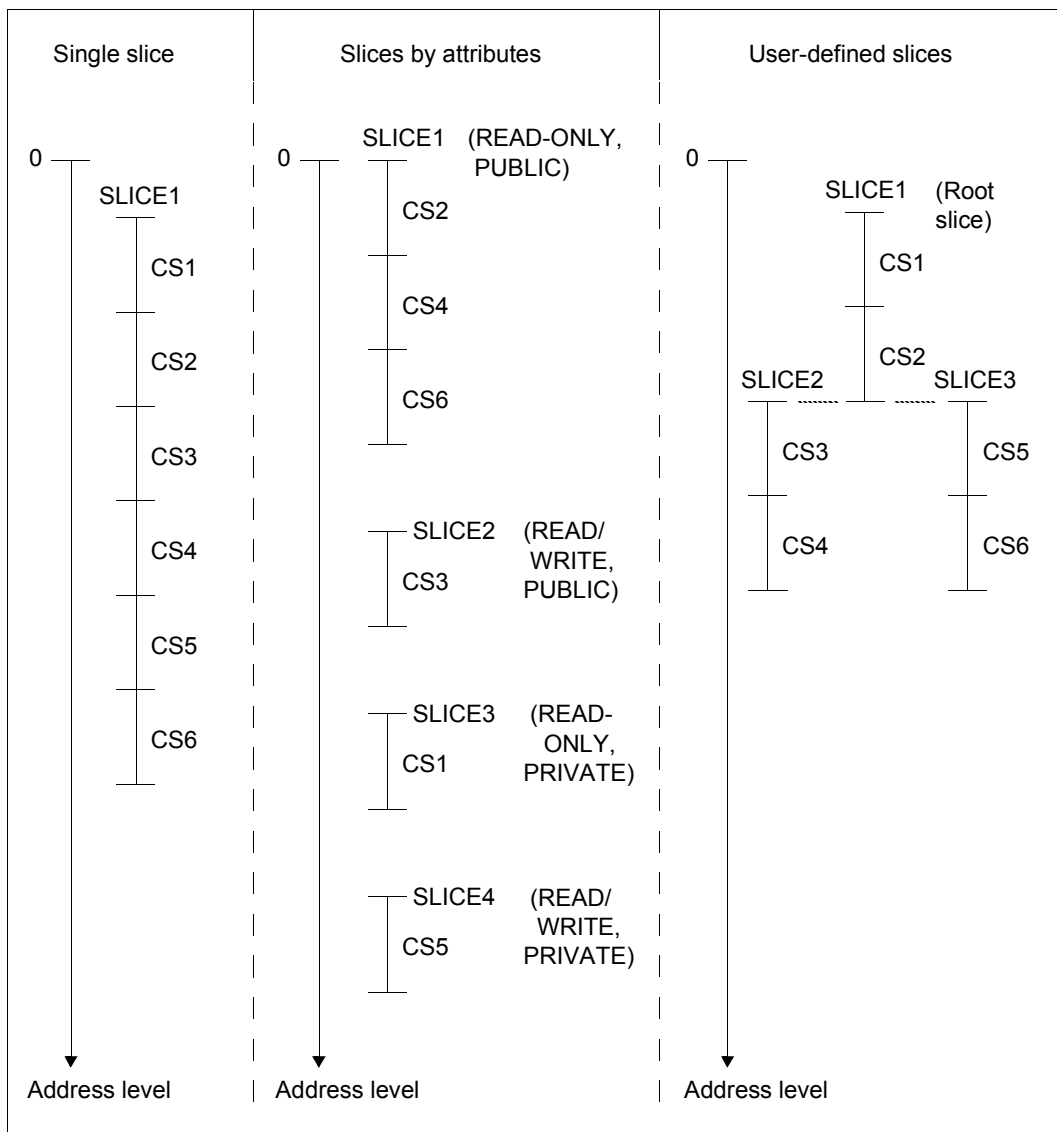


Figure 3: Example of the physical structures of an LLM

## 2.4 Identification of an LLM

Each LLM can be addressed:

- by means of an internal name and an internal version if it is created in the BINDER work area
- by means of an element name and an element version if it is saved as a library element in a program library.

Sub-LLMs are addressed by means of their path name.

### Internal name and internal version

The internal name identifies the root in the tree structure of the LLM (see [page 16](#)). It is specified using the START-LLM-CREATION statement (INTERNAL-NAME operand) and can be modified with the MODIFY-LLM-ATTRIBUTES statement (INTERNAL-NAME operand). If modules are to be replaced in the current LLM (REPLACE-MODULES) or removed (REMOVE-MODULES), the internal names must be used.

In addition to the internal name, an internal version (INTERNAL-VERSION operand) can be specified in the START-LLM-CREATION statement. This can be modified using the MODIFY-LLM-ATTRIBUTES statement (INTERNAL-VERSION operand).

The internal name and the internal version are taken over as element name and element version on saving of the LLM in a program library if corresponding values are set in the SAVE-LLM statement.

The internal name and the internal version are logged with a message when the LLM is loaded.

### Element name and element version

In a program library a library element is identified by the **element type** and the **element identifier** (see the “LMS” manual [3]). For an LLM stored as an element in a program library, the element type is always “L”. The element identifier is composed of the element name and the element version of the LLM.

The element name and the element version are specified in the SAVE-LLM statement (ELEMENT and VERSION operands).

**Path name**

Sub-LLMs and thus also the OMs within a sub-LLM are addressed through their path names. The path name of a sub-LLM or OM consists of a hierarchically organized sequence of individual names separated from one another by a period. The path name has the following format:

:<pathname>: = internal-name.subLLM-level-1.subLLM-level-2. ... . subLLM-level-n

The character string '...' must be replaced by:

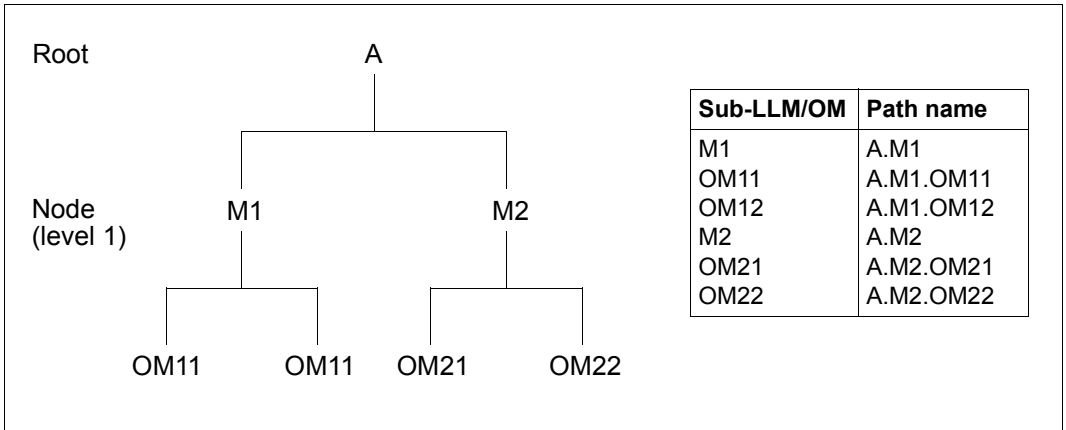
- a sequence of further sub-LLM levels separated by periods or
- an empty character string (see “Abbreviation options”, [page 27](#)).

The structure of the path name is identical with the logical structure of the LLM (see [page 16ff](#)). This means:

- The first name “internal-name” is always the internal name (root) of the LLM tree.
- The last name “subLLM-level-n” identifies the node to be addressed (level n).
- The names between the first and last identify the nodes that lie between the root and the last level n. They form the link between the first and last names.

*Example*

LLM with node level 1





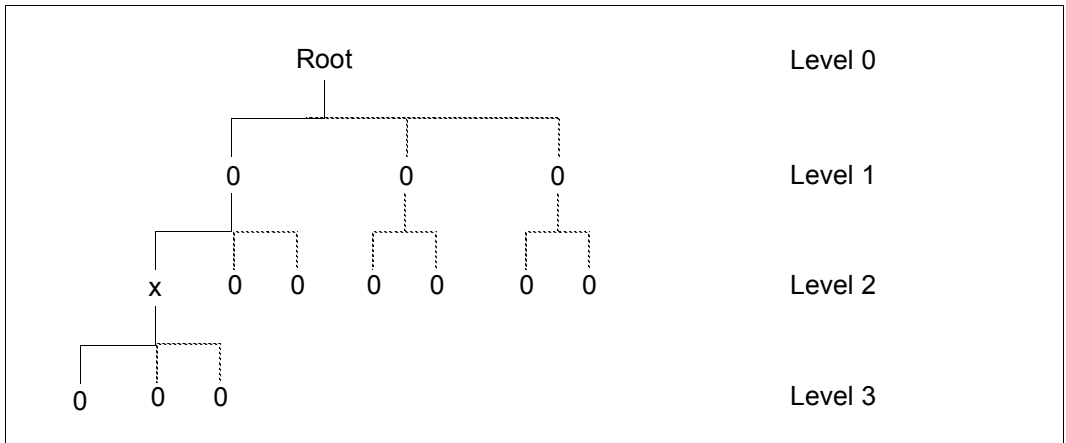
### Path name abbreviation

The user has the option of abbreviating the path name for a desired OM.

#### *Description of the search procedure*

In order to reach a desired OM, BINDER uses a method known as “backtracking” when searching for the OM.

Starting from the root, the search proceeds along the leftmost path. When BINDER reaches a branching point, the leftmost path is again selected.

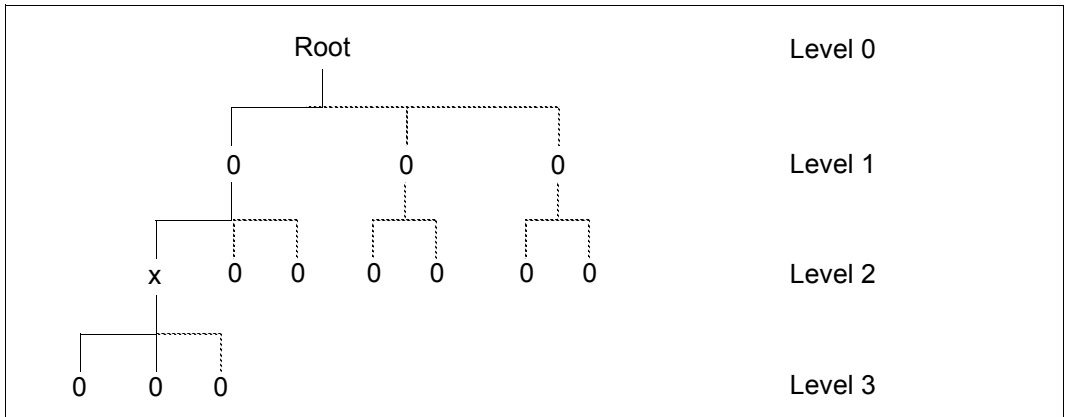


Key to example and the following:

- O Any node
- Path already searched
- ..... Path not yet searched
- x,y.. Node designation

When BINDER reaches level n (lowest level in the LLM, level 3 in the example) without finding the desired OM, it returns to the node at level n-1 (node x on level 2 in the example). This procedure is known as “backtracking”.

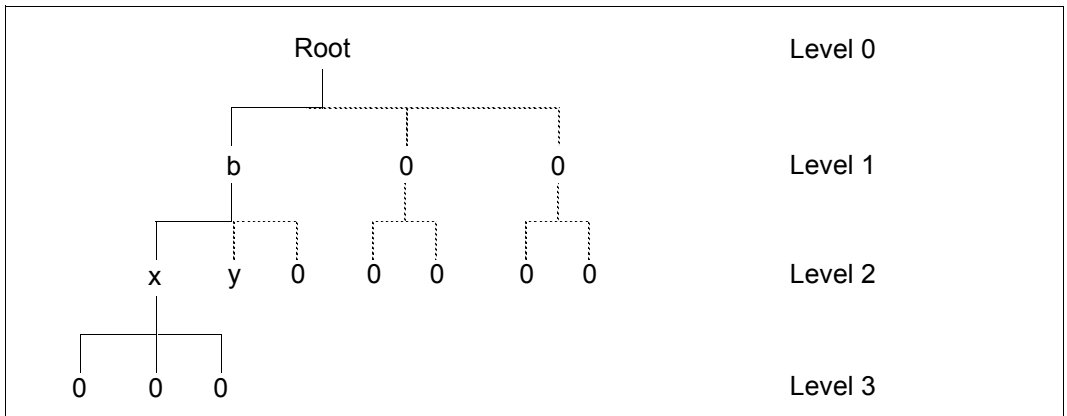
Here BINDER checks whether any other paths branch off from this node. If this is the case, BINDER continues the search on the path nearest to the one previously searched (“second leftmost” path).



This process is repeated until either the desired OM is found or all branches of a node have been examined.

When all branches of a node have been searched without finding the OM, BINDER backtracks another level and examines the next branch of this higher-level node.

In the example, BINDER backtracks to node b on level 1 and continues its search on the path from node b to node y. If node y is identical with the desired OM, the search is successfully terminated.



If node y is not identical with the desired OM, the third path of node b is examined. If the search on this path is also unsuccessful, BINDER backtracks to the root and then examines the center sub-LLM in the same manner.

*Abbreviation options*

If only one OM of the same name exists in the entire LLM, the path name may be abbreviated as follows:

```
:<pathname>: = .subLLM-level-n
```

If two or more OMs of the same name exist in the LLM, the path name must contain at least one intermediate node in order to uniquely define the path to the desired OM for BINDER.

The following formats are possible:

a)

```
:<pathname>: = internal-name..subLLM-level-n
```

b)

```
:<pathname>: = internal-name.subLLM-level-1..subLLM-level-n
```

Re format a):

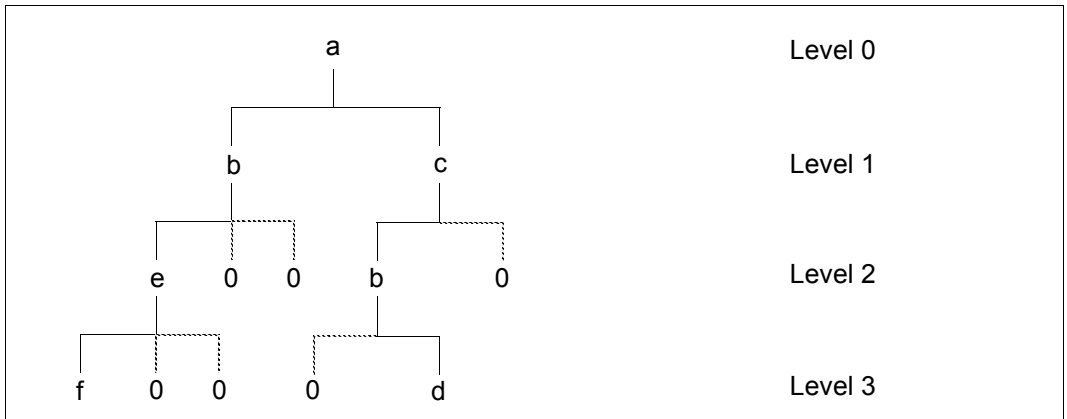
If subLLM-level-n = subLLM-level-1 in this format, the two periods in the path name are replaced by a single period, i.e.

internal-name..subLLM-level-1 = internal-name.subLLM-level-1.

This may lead to errors if

- a node on the first level and a node on a lower level in another path further to the right have the same name and
- the desired node that is not a first-level node is located in the path lying further to the right.

*Example*



Desired node	Possible path specifications		Search successful
	abbreviated	complete	
f	.f	a.b.e.f	yes
	.b..f	a.b.e.f	yes
	a.b..f	a.b.e.f	yes
d	.d	a.c.b.d	yes
	a.c..d	a.c.b.d	yes
	a..b.d		no

Specifying “a..b.d” in this example would result in an error since

- the path “a..b.d” is converted into the path “a.b.d”
- d is not a node in the left subtree,

The path conversion is performed here because node a is a direct successor of root a in the left subtree.

Re format b):

Depending on the position of the desired node, this format can be extended through the specification of further intermediate nodes.

## 2.5 Contents of an LLM

An LLM always contains at least the **text information** and the **physical structure information**.

- **Text information (TXT)**

The text information consists of the code and the data of the modules.

- **Physical structure information**

The physical structure information contains the description of the slices which belong to the LLM.

The user can specify whether the LLM is also to contain the following information:

- **External Symbols Vector (ESV)**

The External Symbols Vector (ESV) contains all program definitions and references. An LLM without an ESV cannot be included or modified by BINDER.

- **Local relocation dictionary (LRLD)**

The local relocation dictionary (LRLD) determines how addresses are aligned (relocated) to a common reference address during linking and loading. If the LRLD is complete, the LLM can be loaded at any desired address. If, however, the LRLD exists only for unresolved external references, the LLM must be loaded at a specific address. However, external references can be resolved. If no local relocation dictionary exists, addresses cannot be relocated and the LLM must be loaded at a specific address. An LLM without a complete LRLD cannot be included or modified by BINDER.

- **Logical structure information**

The logical structure information contains information about the logical structure of the LLM. The logical structure information of an LLM can be present in its entirety (LLM with all sub-LLMs and object modules) or only in part (only object modules (leaves), no sub-LLMs). If no logical structure information is present, the LLM cannot be included or modified by BINDER.

- **List for symbolic debugging (LSD)**

The list for symbolic debugging (LSD) is required by the debugging and diagnostic aids for testing on the source language level. This presupposes that corresponding compiler options are set during compilation of the source program.

The LSD information can be stored only if the External Symbols Vector (ESV) is also stored. If no list for symbolic debugging is present, testing on the source language level is not possible.

- **Descriptors for initialization and termination routines (“Ini/Fini” information)**

The descriptors for initialization and termination routines are required by new compilers for object-oriented programming languages. They allow all the initialization and termination routines of an LLM to be executed in a defined order before or after the actual module code. They have the following contents:

- type of routine: initialization or termination
- information about the address of the routine
- an identifier, which is generated by the compiler

If an LLM contains “Ini/Fini” information, this is noted in the LLM’s logical root node.

BLSSERV is required to load LLMs that contain “Ini/Fini” information. Therefore, they cannot be loaded in BS2000 versions earlier than V3.0, and only in BS2000 version V3.0, if at least version V2.0 of BLSSERV is used.

The compiler determines whether or not the LLM contains “Ini/Fini” information.

Whether or not the LLM is to contain the External Symbols Vector (ESV) and/or the local relocation dictionary (LRLD) is defined by the user on saving the LLM with the SAVE-LLM statement.

Logical structure information and/or LSD information can be selected by the user when:

- creating an LLM (START-LLM-CREATION)
- updating an LLM (START-LLM-UPDATE)
- modifying the attributes of an LLM (MODIFY-LLM-ATTRIBUTES)
- saving an LLM (SAVE-LLM)
- including modules (INCLUDE-MODULES)
- replacing modules (REPLACE-MODULES)
- resolving external references by autolink (RESOLVE-BY-AUTOLINK)

The user can ascertain which additional information an LLM contains from the lists output by the SHOW-MAP statement or on saving the LLM with the SAVE-LLM statement (see [page 141ff](#)).

## 2.6 Limiting conditions for LLMs

Specifying the attributes and contents of an LLM restricts the set of actions which can subsequently be executed. For this reason, certain limiting conditions must be observed when storing an LLM.

The table below lists the values of various operands in the SAVE-LLM statement and the consequences which result from these for subsequent processing.

<i>Operand</i>	REQUIRED-COMPRESSION
<i>Value</i>	YES
<i>Result</i>	The LLM can be loaded in all BS2000 versions.

<i>Operand</i>	LOGICAL-STRUCTURE	SYMBOL-DICTIONARY	RELOCATION-DATA
<i>Value</i>	NONE	NO	UNRESOLVED-ONLY or NO
<i>Result</i>	<ol style="list-style-type: none"> <li>1. The LLM cannot be modified later.</li> <li>2. The LLM cannot be included in another LLM.</li> </ol>		

<i>Operand</i>	RELOCATION-DATA
<i>Value</i>	UNRESOLVED-ONLY or NO
<i>Result</i>	The LLM is not relocatable since the addresses cannot be relocated. If the LLM is to be relocatable, RELOCATION-DATA=YES must be specified.
<i>Value</i>	NO
<i>Result</i>	Unresolved external references cannot be resolved. They can be resolved only if RELOCATION-DATA = YES or RELOCATION-DATA = UNRESOLVED-ONLY is specified.

<i>Operand</i>	TEST-SUPPORT
<i>Value</i>	NO
<i>Result</i>	Testing at the source language level (e.g. with the debugger AID; see the "AID (BS2000)" manual [8]) is not possible.

It should also be noted that an LLM with user-defined slices *cannot* be included in another LLM. However, it is possible to modify an LLM with user-defined slices.

### LLM format

The table below shows the dependencies between the LLM format, the value of the FOR-BS2000-VERSIONS operand with which it is selected, and the DBL and BLSSERV versions with which it can be processed:

LLM format	FOR-BS2000-VERSIONS operand	Loadable with
1	FROM-V10	DBL BS2000 V10.0A or higher
2	FROM-OSD-V1	DBL BS2000/OSD-BC V1.0A or higher
3	FROM-OSD-V3	DBL BS2000/OSD-BC V3.0A or higher <sup>1</sup>
4	FROM-OSD-V4	BLSSERV V2.0A BS2000/OSD-BC V4.0A or higher

<sup>1</sup> An LLM format 3 containing X86 code can be loaded with BS2000/OSD-BC V9.0 or higher.



There are also dependencies between the LLM format and some LLM attributes. The table below shows which LLM format is required for which LLM attributes:

Attribute	LLM format
REQUIRED-COMPRESSION=*YES	≥ 2
CONNECTION-MODE=*BY-RESOLUTION	1, 3, 4
RISC code present	≥ 3 <sup>1</sup>
RESOLUTION-SCOPE defined	≥ 3 <sup>2</sup>
LLM output in PAM file	≥ 3
EEN names (see <a href="#">section “Symbol names” on page 106</a> )	4 <sup>3</sup>
Initialization/termination information	4 <sup>4</sup>

<sup>1</sup> If the operand RELOCATION-DATA=\*NO has been specified for SAVE-LLM, the LLM can also be saved in format 1 or 2

<sup>2</sup> The LLM can be saved in format 1 or 2, but cannot be further processed with BINDER. The RESOLUTION-SCOPE specification is ignored during the save.

<sup>3</sup> The LLM can be saved in format 1 - 3 if the EEN names (see [section “Symbol names” on page 106](#)) are suppressed, but cannot be further processed with BINDER. The EEN names can be suppressed in one of the following ways:  
with the SYMBOL-DICTIONARY operand or automatically if all EEN external references are resolved and FOR-BS2000-VERSION ≠ FROM-OSD-V4. The EEN names (see [section “Symbol names” on page 106](#)) are not saved in this case.

<sup>4</sup> The LLM can be saved in format 1 - 3, but cannot be further processed with BINDER. The “Ini/Fini” information is saved, but not the information that the LLM contains “Ini/Fini” information.

If one of the LLM attributes listed above requires a format with a number higher than specified with the FOR-BS2000-VERSIONS operand, BINDER outputs an error message.



---

## 3 BINDER functions

The BINDER functions are divided into the following function groups:

- creating, modifying and saving an LLM
- including, removing and replacing modules
- creating the logical structure of an LLM
- creating the physical structure of an LLM
- resolving external references
- handling symbols
- merging modules
- changing the attributes of LLMs and modules
- display functions
- controlling list output and error processing

Figure 4 illustrates the function groups with the associated BINDER statements and how they interact.

The individual function groups are described in detail in the following sections.

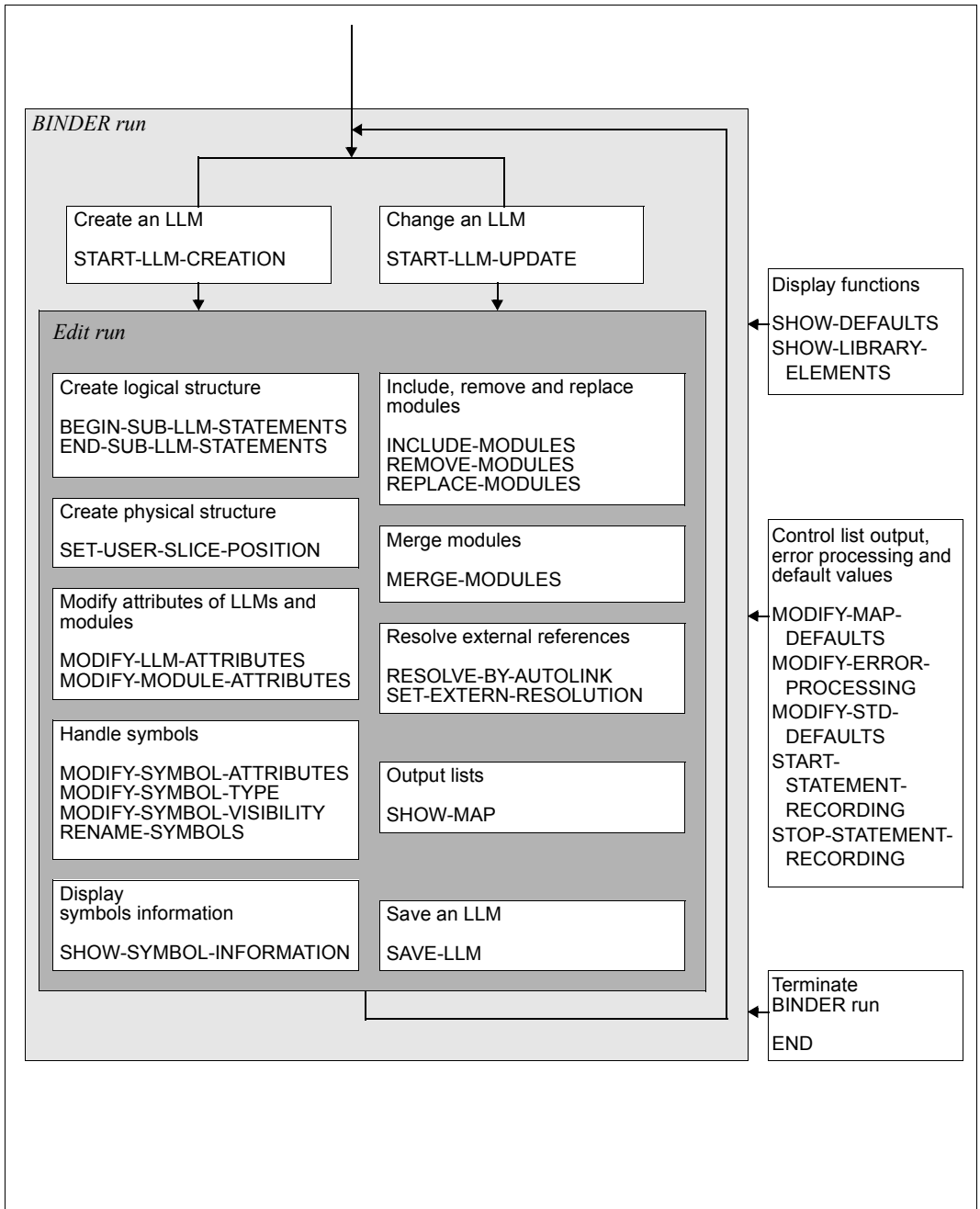


Figure 4: Overview of the BINDER functions

## 3.1 Creating, modifying and saving an LLM

### 3.1.1 Creating an LLM

An LLM is created in the **work area** of BINDER by means of the START-LLM-CREATION statement. This involves definition of the attributes of the LLM in accordance with the information in the START-LLM-CREATION statement. The LLM created in the BINDER work area is referred to as the **current LLM**. The current LLM can then be processed in the work area. For example,

- modules can be included (INCLUDE-MODULES)
- modules can be removed (REMOVE-MODULES)
- modules can be replaced (REPLACE-MODULES).

Processing of the current LLM is terminated without implicit saving of the LLM. It is saved as a type L element in a program library by means of the SAVE-LLM statement.

If an element with the same element name and the same element version already exists in the program library, it will be overwritten if OVERWRITE=YES is specified.

When creating an LLM, the internal name (INTERNAL-NAME) must be specified in the START-LLM-CREATION statement (see [page 138f](#)). It is entered as the element name for the LLM in the program library if corresponding values were selected for the element name in the SAVE-LLM statement on saving the LLM (ELEMENT=\*INTERNAL-NAME).

The following optional specifications may be made:

- the internal version (INTERNAL-VERSION)  
This is entered as the element version for the LLM in the program library if corresponding values were selected for the element version in the SAVE-LLM statement on saving the LLM (VERSION=\*INTERNAL-VERSION). The element version is logged by means of a message on loading the LLM.
- the physical structure (SLICE-DEFINITION)  
LLMs with single slices, LLMs with slices by attributes or LLMs with user-defined slices can be created (see [page 18ff](#)).
- copyright information (COPYRIGHT)  
This comprises a text and the year number that are entered in the LLM. The copyright information is logged by means of a message on loading the LLM.

- Declarations concerning the logical structure information and the LSD information (INCLUSION-DEFAULTS).



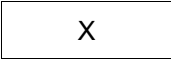

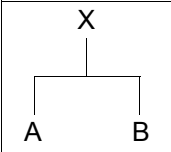

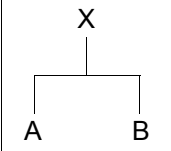
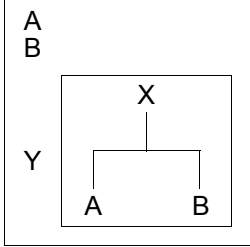
With the LOGICAL-STRUCTURE suboperand the user can define whether the logical structure information is to be taken over from the modules into the current LLM during inclusion or replacement of modules or whether substructures (sub-LLMs) are to be ignored. In the latter case, only the *object modules (OMs)* are taken over from the modules during inclusion or replacement of modules.

With the TEST-SUPPORT suboperand the user can define whether the LSD (list for symbolic debugging) information is to be taken over from the modules during inclusion, removal or replacement of modules.

The values of the LOGICAL-STRUCTURE and TEST-SUPPORT operands remain in force up to the next START-LLM-CREATION or START-LLM-UPDATE statement (see edit run, [page 138](#)). They can be used as default values in all INCLUDE-MODULES, RESOLVE-BY-AUTOLINK and REPLACE-MODULES statements. The values can be changed by means of the MODIFY-LLM-ATTRIBUTES statement.

*Example*

Creating and saving an LLM

<u>Statements</u>	<u>Current LLM</u> (work area)	<u>Program library</u> (LIB1)
(1) START-BINDER		
(2) START-LLM-CREATION INTERNAL-NAME=X		
(3) INCLUDE-MODULES LIBRARY=LIB1, ELEMENT=(A,B)		
(4) SAVE-LLM LIBRARY=LIB1, ELEMENT=Y		
(5) END		

Explanation:

- (1) BINDER call. BINDER sets up a work area.
- (2) An LLM with the internal name X is created in the work area. The internal name X forms the root in the logical structure of the LLM.
- (3) The object modules A and B are read from program library LIB1 and included in the current LLM.
- (4) The LLM created is saved as an element with the element name Y in program library LIB1.
- (5) End of BINDER run.

### 3.1.2 Updating an LLM

An LLM saved as a type L element in a program library is updated by means of the START-LLM-UPDATE statement. The LLM is read from the program library into the BINDER work area during this process. When the LLM has been read in it becomes the **current LLM**, i.e. it has the same status as before it was saved in the program library. The current LLM can then be processed in the work area. For example,

- modules can be included (INCLUDE-MODULES)
- modules can be removed (REMOVE-MODULES)
- modules can be replaced (REPLACE-MODULES)
- attributes of the LLM can be modified (MODIFY-LLM-ATTRIBUTES)
- attributes of modules can be modified (MODIFY-MODULE-ATTRIBUTES).

With the LOGICAL-STRUCTURE suboperand the user can define whether the logical structure information is to be taken over from the modules into the current LLM during inclusion, removal or replacement of modules or whether substructures (sub-LLMs) are to be ignored. In the latter case, only the *object modules (OMs)* are taken over from the modules during inclusion or replacement of modules.

With the TEST-SUPPORT suboperand the user can define whether the LSD (list for symbolic debugging) information is to be taken over from the modules during inclusion or replacement of modules.

The values assumed as default values for the LOGICAL-STRUCTURE and TEST-SUPPORT operands are those defined on creation of the LLM with the START-LLM-CREATION statement.

The values of the LOGICAL-STRUCTURE and TEST-SUPPORT operands remain in force up to the next START-LLM-CREATION or START-LLM-UPDATE statement (see edit run, [page 138](#)). They can be used as default values in all INCLUDE-MODULES, RESOLVE-BY-AUTOLINK and REPLACE-MODULES statements. The default values for LOGICAL-STRUCTURE and TEST-SUPPORT can be changed with the MODIFY-LLM-ATTRIBUTES statement.

Processing of the current LLM is terminated without implicit saving of the LLM. It is saved as a type L element in a program library by means of the SAVE-LLM statement. If the new element keeps the same element name and the same element version, the previous element will be overwritten in the program library if OVERWRITE=YES is specified.



*Example*

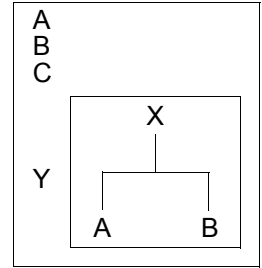
Updating and saving an LLM

Statements

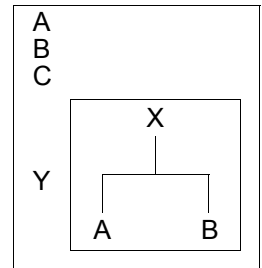
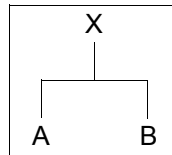
Current LLM  
(work area)

Program library  
(LIB1)

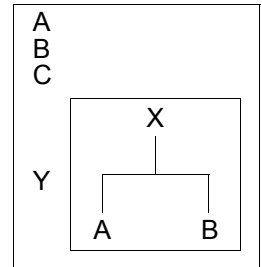
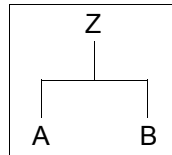
(1) START-BINDER



(2) START-LLM-UPDATE  
LIBRARY=LIB1  
ELEMENT=Y



(3) MODIFY-LLM-ATTRIBUTES  
INTERNAL-NAME=Z

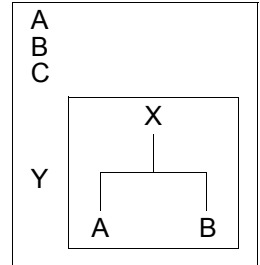
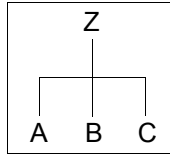


Statements

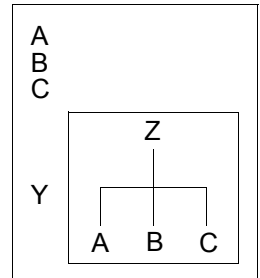
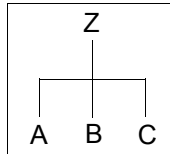
Current LLM  
(work area)

Program library  
(LIB1)

(4) INCLUDE-MODULES  
ELEMENT=C



(5) SAVE-LLM  
ELEMENT=Y,  
OVERWRITE=YES



(6) END

Explanation:

- (1) BINDER call. BINDER sets up a work area.
- (2) An LLM saved as an element with the element name Y in program library LIB1 is to be updated. To this end the LLM is read into the work area.
- (3) The internal name X of the LLM is modified. The new internal name is Z.
- (4) The object module C is read from program library LIB1 and included in the LLM. The library name LIB1 is taken over from the preceding START-LLM-UPDATE statement (current program library).
- (5) The updated LLM is saved as an element with the *same* element name Y in program library LIB1, overwriting the previous element.
- (6) End of BINDER run.

### 3.1.3 Modifying the attributes of an LLM

Attributes of an LLM defined during creation of the LLM with the START-LLM-CREATION statement can be modified by means of the MODIFY-LLM-ATTRIBUTES statement. The statement modifies the following attributes:

- the internal name (INTERNAL-NAME)
- the internal version (INTERNAL-VERSION)
- the type of physical structure of the LLM (SLICE-DEFINITION)
- the copyright information (COPYRIGHT)
- declarations concerning logical structure information (LOGICAL-STRUCTURE)
- declarations concerning LSD information (TEST-SUPPORT).

The type of the physical structure of the LLM may be modified as follows:

1. LLM with slices by attributes → LLM with single slice
2. LLM with single slice → LLM with slices by attributes
3. LLM with slices by attributes → LLM with slices by *other* attributes
4. LLM with user-defined slices → LLM with user-defined slices and modified values for AUTOMATIC-CONTROL and EXCLUSIVE-SLICE-CALL.

### 3.1.4 Saving an LLM

The current LLM created or updated in the work area by means of a START-LLM-CREATION or START-LLM-UPDATE statement, respectively, is saved as a type L element in a program library by means of the SAVE-LLM statement. An LLM updated with the START-LLM-UPDATE statement normally overwrites the existing element if the new element keeps the same element name.

However, the OVERWRITE=NO operand can be used to prevent overwriting of the existing element: the user then receives an error message if he/she attempts to save an element under the same name and with the same version number. In contrast, if OVERWRITE=YES is specified and the element does not already exist, the user does not receive an error message.

When saving the LLM you can either explicitly specify the name of the program library or select the **current** program library. The current program library is the library to which the most recent preceding START-LLM-UPDATE or SAVE-LLM statement related.

The element name and element version that the LLM is to receive when saved in the program library can either be specified explicitly or the current name and the current version are assumed. The current name and current version are taken by BINDER from the most recent SAVE-LLM statement specified after the most recent START-LLM-CREATION or START-LLM-UPDATE statement.

If no corresponding SAVE-LLM statement has been specified, BINDER takes as the element name and element version

- the element name and element version from the most recent START-LLM-UPDATE statement or
- the *internal name* and *internal version* from the most recent START-LLM-CREATION statement.

The FOR-BS2000-VERSIONS operand can be used to specify the BS2000 version in which the LLM is to be loadable with DBL.

If FOR-BS2000-VERSIONS=\*FROM-V10 is specified, the LLM is stored in *format 1*.

If FOR-BS2000-VERSIONS=\*FROM-OSD-V1 is specified, the LLM is stored in *format 2* provided the user specified REQUIRED-COMPRESSION=YES or the LLM contains slices created with the attribute PUBLIC.

LLMs in format 2 have the advantage that their text information can be compressed and that resolution of the external references in the private slice by the public slice provides better performance.

#### Notes

The operand REQUIRED-COMPRESSION can be used to compress the text information (TXT) in LLMs.

The handling of name conflicts is controlled with the operand NAME-COLLISION (see also [page 102ff](#)).

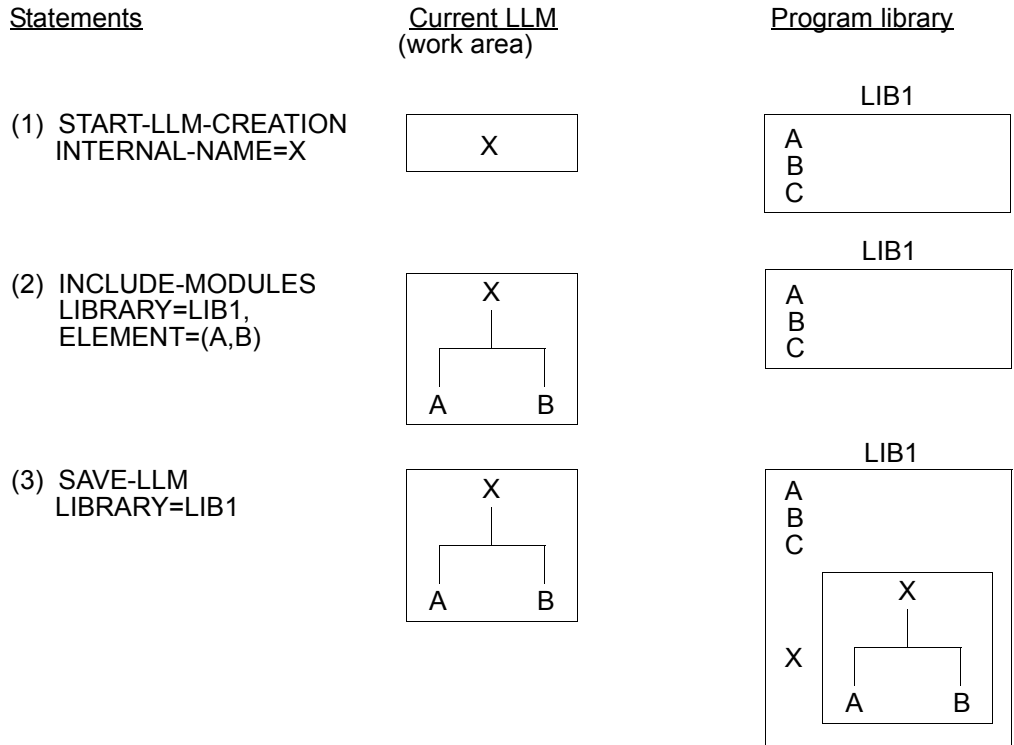
The default values for the operands FOR-BS2000-VERSIONS, OVERWRITE, REQUIRED-COMPRESSION and NAME-COLLISION can be modified with the MODIFY-STD-DEFAULTS statement.

When saving the LLM, the user can define the additional information (see [page 29f](#)) that is to be included. However, certain limiting conditions (see [page 31f](#)) should be noted when doing this. The following additional information may be selected:

- External Symbols Vector (ESV) (SYMBOL-DICTIONARY operand)
- local relocation dictionary (LRLD) (RELOCATION-DATA operand)
- logical structure information (LOGICAL-STRUCTURE operand) and
- LSD (list for symbolic debugging) information (TEST-SUPPORT operand).

*Example*

Saving an LLM in different program libraries

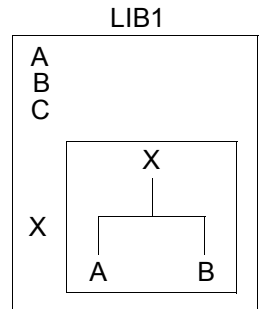
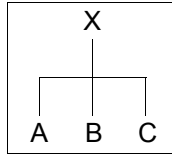


Statements

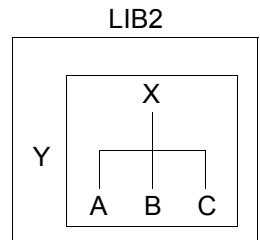
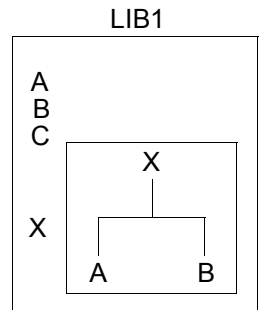
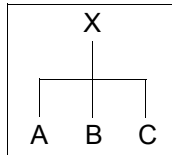
Current LLM  
(work area)

Program library

(4) INCLUDE-MODULES  
ELEMENT=C



(5) SAVE-LLM  
LIBRARY=LIB2,  
ELEMENT=Y



Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) Object modules A and B are read from program library LIB1 and included in the current LLM.
- (3) The current LLM is saved in program library LIB1. The internal name X from the most recent START-LLM-CREATION statement is taken over as the element name.
- (4) Object module C is read from program library LIB1 and included in the current LLM. The library name LIB1 is taken over from the most recent preceding INCLUDE-MODULES statement.
- (5) The current LLM is saved in program library LIB2 as an element with the element name Y.

## 3.2 Including, removing and replacing modules

### 3.2.1 Including modules

With the INCLUDE-MODULES statement, BINDER includes modules in the current LLM in the work area.

Both object modules (OMs) and LLMs can be included as modules. However, LLMs with user-defined slices and LLMs without relocation data, without logical structure information or without an External Symbols Vector cannot be included.

Whole LLMs can be included, or sub-LLMs selected, if the complete structure information was included when saving the LLM (LOGICAL-STRUCTURE=WHOLE-LLM operand).

Sub-LLMs are selected by means of their path name (operand ELEMENT=...(...,SUB-LLM=...)...).

The slice structure of the input LLM has no effect on the generation of slices in the LLM currently being processed.

The following input sources may be used:

- for object modules: a program library (element type R), an object module library (OML) or the EAM object module file (OMF)
- for LLMs and sub-LLMs: a program library (element type L).

The input source can either be specified explicitly or the current input source can be taken over. The current input source is the library or EAM object module file from which the last module was taken (by means of a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES) statement.

All modules or individual explicitly specified modules can be included in the LLM from the selected input source.

It is possible to specify for the modules that only LLMs, only OMs, or both types are to be searched for. Should identical names exist when both LLMs *and* OMs are being sought in a program library, the priority can be defined through the TYPE operand. By default, an LLM has a higher priority than an OM.

Modules in a program library are selected on the basis of their element version. If no element version has been explicitly specified, the element with the *highest* element version is assumed (see the “LMS” manual [3]). The name of the logical node (NAME operand) generated by including a module can be either the internal name of the module (LLM/OM), the external name of the module or a name assigned by the user.

With the RUN-TIME-VISIBILITY operand, the user can specify whether or not a module is to be regarded as a runtime module. If RUN-TIME-VISIBILITY=YES is specified, all symbols of this module are masked when the LLM is stored, but previously resolved external references remain resolved.



These symbols are again made visible, for resolution of external references when including or updating this LLM, during the BINDER run.

When including modules, the user can also control the handling of name conflicts.

*Example 1*

Different LLMs are created in succession in the work area. LLMs and OMs taken from program library LIB1 are included in the current LLM. The current LLM is saved in program library LIB1.

The program library LIB1 with the file link name EXLINK contains the following modules:

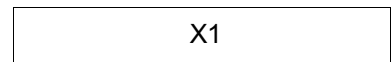
OMs: A,B

LLMs: A,B,Y,Z

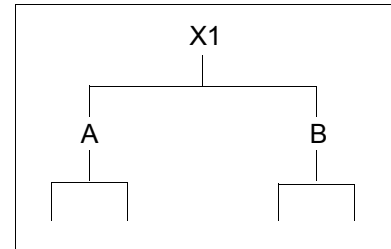
Statements

Current LLM  
(work area)

(1) START-LLM-CREATION  
INTERNAL-NAME=X1

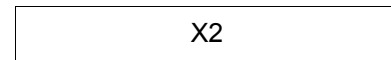


(2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B)

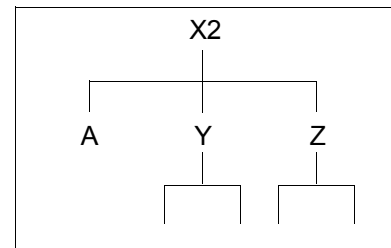


(3) SAVE-LLM LIBRARY=LIB1

(4) START-LLM-CREATION  
INTERNAL-NAME=X2

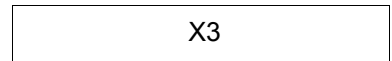


(5) INCLUDE-MODULES  
LIBRARY=\*LINK(EXLINK),  
ELEMENT=(A,Y,Z),  
TYPE=(R,L)

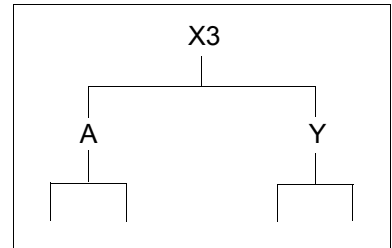


(6) SAVE-LLM LIBRARY=LIB1

(7) START-LLM-CREATION  
INTERNAL-NAME=X3

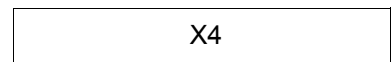


(8) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,Y),  
TYPE=L

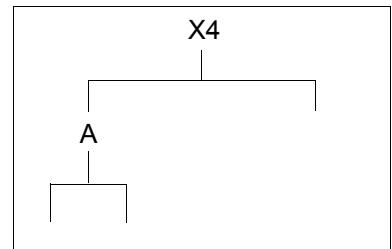


(9) SAVE-LLM LIBRARY=LIB1

(10) START-LLM-CREATION  
INTERNAL-NAME=X4

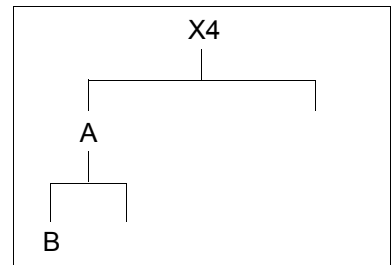


(11) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=A,TYPE=L



(12) INCLUDE-MODULES ELEMENT=B,  
TYPE=R, PATH-NAME=X4.A

(13) SAVE-LLM LIBRARY=LIB1



Explanation:

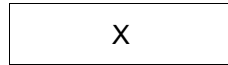
- (1) An LLM with the internal name X1 is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1 since, by default, in the case of elements having identical names a higher priority is defined for LLMs.
- (3) The current LLM is saved in program library LIB1. The internal name X1 from the most recent START-LLM-CREATION statement is taken over as the element name.
- (4) An LLM with the internal name X2 is created in the work area.
- (5) From program library LIB1 referenced with the file link name EXLINK, the following modules are included in the current LLM:
  1. object module A since the TYPE=(R,L) operand defines a higher priority for object modules in the event of elements having identical names.
  2. LLMs Y and Z.
- (6) The current LLM is saved in program library LIB1. The internal name X2 from the most recent START-LLM-CREATION statement is taken over as the element name.
- (7) An LLM with the internal name X3 is created in the work area.
- (8) From program library LIB1, the LLMs A and Y are included in the current LLM since only LLMs are selected through the TYPE=L operand.
- (9) The current LLM is saved in program library LIB1. The internal name X3 from the most recent START-LLM-CREATION statement is taken over as the element name.
- (10) An LLM with the internal name X4 is created in the work area.
- (11) From program library LIB1, the LLM A is included since only LLMs are selected through the TYPE=L operand.
- (12) Object module B is included as a sub-LLM on the node having the path name X4.A since only object modules are selected through the TYPE=R operand. The program library LIB1 is defined by the preceding INCLUDE-MODULES statement (current program library).
- (13) The current LLM is saved in program library LIB1. The internal name X4 from the most recent START-LLM-CREATION statement is taken over as the element name.

*Example 2*

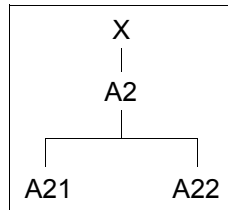
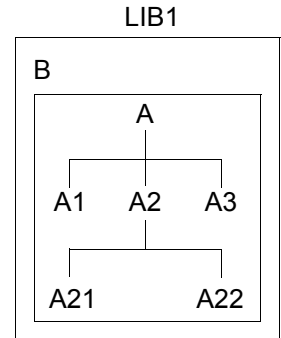
Including a sub-LLM

Statements

(1) START-LLM-CREATION  
INTERNAL-NAME=X

Current LLM  
(work area)

(2) INCLUDE-MODULES  
LIBRARY=LIB1,  
ELEMENT=B  
(SUB-LLM=A.A2)

Program library

Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) A sub-LLM A2 of an LLM A that is saved as element B in program library LIB1 is included in the current LLM. The sub-LLM A2 is referenced in the LLM A through the path name A.A2.

### 3.2.2 Removing modules

BINDER removes modules from the current LLM with the REMOVE-MODULES statement. Object modules and sub-LLMs are removed. The sub-LLMs are defined through their path names.

The following are not removed:

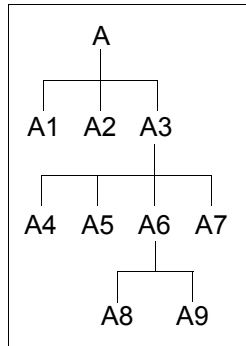
- the current sub-LLM (see [page 57ff](#))
- a sub-LLM whose beginning is defined with the BEGIN-SUB-LLM-STATEMENTS statements but whose end has not yet been specified by means of the END-SUB-LLM-STATEMENTS statement, and any sub-LLM containing a sub-LLM as described here (see [page 57ff](#)).

*Example*

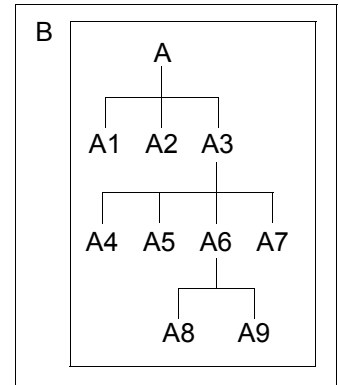
Removing an object module and a sub-LLM

Statements

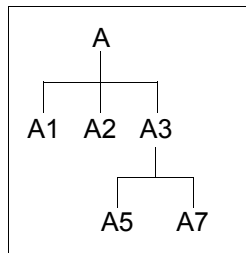
(1) START-LLM-UPDATE  
LIBRARY=LIB1,  
ELEMENT=B

Current LLM  
(work area)Program library

LIB1



(2) REMOVE-MODULES  
NAME=(A4,A6),  
PATH-NAME=A.A3



Explanation:

- (1) An LLM saved as an element with the name B in program library LIB1 is to be updated. To this end the LLM is read into the work area.
- (2) Object module A4 and sub-LLM A6, contained in sub-LLM A3 of the current LLM, are removed. The path name PATH-NAME=A.A3 references the sub-LLM A3.

### 3.2.3 Replacing modules

The REPLACE-MODULES statement replaces modules in the current LLM. The modules in the current LLM can be replaced by object modules, LLMs, or both. Since replacing modules involves implicit inclusion of other modules, the same restrictions apply as for including modules (see [page 48](#)).

The following input sources may be used:

- for object modules: a program library (element type R), an object module library (OML) or the EAM object module file (OMF)
- for LLMs and sub-LLMs: a program library (element type L).

The input source can either be specified explicitly or the current input source can be taken over. The current input source is the library or EAM object module file from which the last module was taken (by means of a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES) statement.

All modules or individual explicitly specified modules can be fetched from the selected input source.

It is possible to specify for the modules that only LLMs, only OMs, or both types are to be searched for. Should identical names exist when both LLMs *and* OMs are being sought in a program library, the priority can be defined through the TYPE operand. Generally an LLM has a higher priority than an OM.

The name of the new logical node (NAME operand) generated by including a module can be either the internal name of the module (LLM/OM), the external name of the module or a name assigned by the user. Modules in a program library are selected on the basis of their element version. If no element version has been explicitly specified, the element with the *highest* element version is assumed (see the “LMS” manual [3]).

With the RUN-TIME-VISIBILITY operand, the user can specify whether or not a module is to be regarded as a runtime module. If RUN-TIME-VISIBILITY=YES is specified, all symbols of this module are masked when the LLM is stored, but previously resolved external references remain resolved. These symbols are again made visible, for resolution of external references when including or updating this LLM, during the BINDER run.

When replacing modules, the user can also control the handling of name conflicts.

*Example 1*

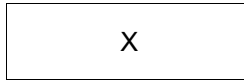
Replacing an object module with an LLM

Statements

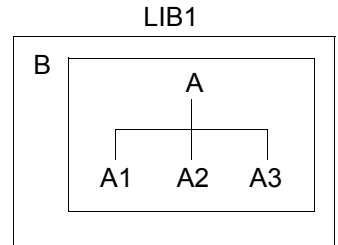
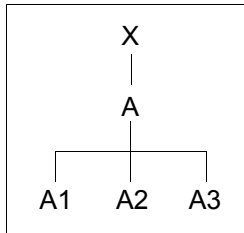
Current LLM  
(work area)

Program library

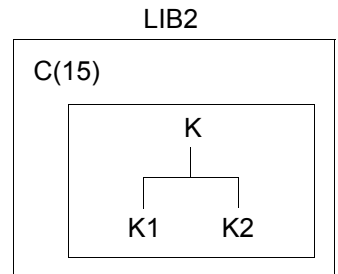
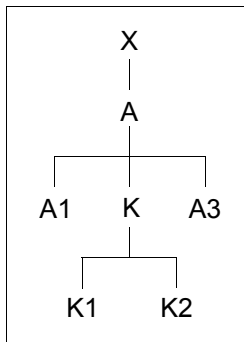
(1) START-LLM-CREATION  
INTERNAL-NAME=X



(2) INCLUDE-MODULES  
LIBRARY=LIB1,  
ELEMENT=B



(3) REPLACE-MODULES  
NAME=A2,  
PATH-NAME=X.A,  
LIBRARY=LIB2,  
ELEMENT=C(15)

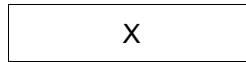
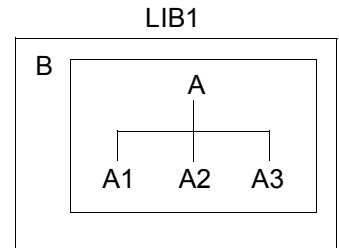
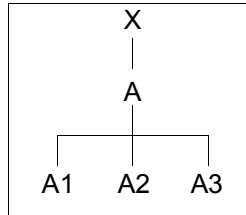
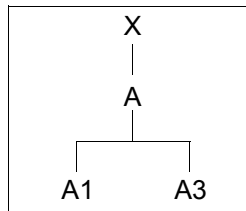
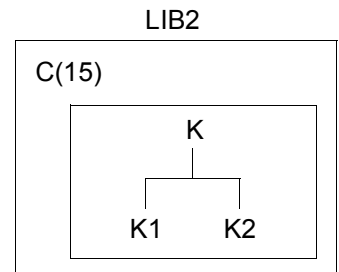
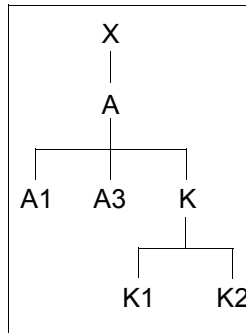


Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The LLM with element name B is included from program library LIB1.
- (3) The LLM with element name C and element version 15 is read from program library LIB2 and replaces object module A2 in the current LLM as a sub-LLM; path name X.A references this sub-LLM.

*Example 2*

Replacing an object module with an LLM

StatementsCurrent LLM  
(work area)Program library(1) START-LLM-CREATION  
INTERNAL-NAME=X(2) INCLUDE-MODULES  
LIBRARY=LIB1,  
ELEMENT=B(3) REMOVE-MODULES  
NAME=A2,  
PATH-NAME=X.A,(4) INCLUDE-MODULES  
LIBRARY=LIB2,  
ELEMENT=C(15),  
PATH-NAME=X.A,

Explanation:

(1)/(2) See Example 1.

(3)/(4) The successive REMOVE-MODULES and INCLUDE-MODULES statements have the same effect as one REPLACE-MODULES statement (see Example 1). However, the structure of the LLM is different from the structure in Example 1.



## 3.3 Creating and modifying the logical structure of an LLM

### 3.3.1 Creating the logical structure of an LLM

The logical structure of an LLM is described by means of nested BEGIN-SUB-LLM-STATEMENTS and END-LLM-STATEMENTS statements. The BEGIN-SUB-LLM-STATEMENTS statement defines the beginning of a sub-LLM, the END-SUB-LLM-STATEMENTS statement the end of a sub-LLM on the corresponding nesting level. INCLUDE-MODULES, REMOVE-MODULES and RESOLVE-BY-AUTOLINK statements, for example, can be specified on any nesting level (see [figure 5](#)).

The node in the logical structure at which the sub-LLM is to begin can be defined through the path name (see [page 24ff](#)) or the current sub-LLM can be taken over.

The current sub-LLM is defined as follows:

- The START-LLM-CREATION or START-LLM-UPDATE statement defines the root of the LLM structure tree as the current sub-LLM.
- Each subsequent BEGIN-SUB-LLM-STATEMENTS statement creates a further level in the current sub-LLM.
- Each END-SUB-LLM-STATEMENTS statement returns the current sub-LLM to the level that contained the current sub-LLM prior to the associated BEGIN-SUB-LLM-STATEMENTS statement.

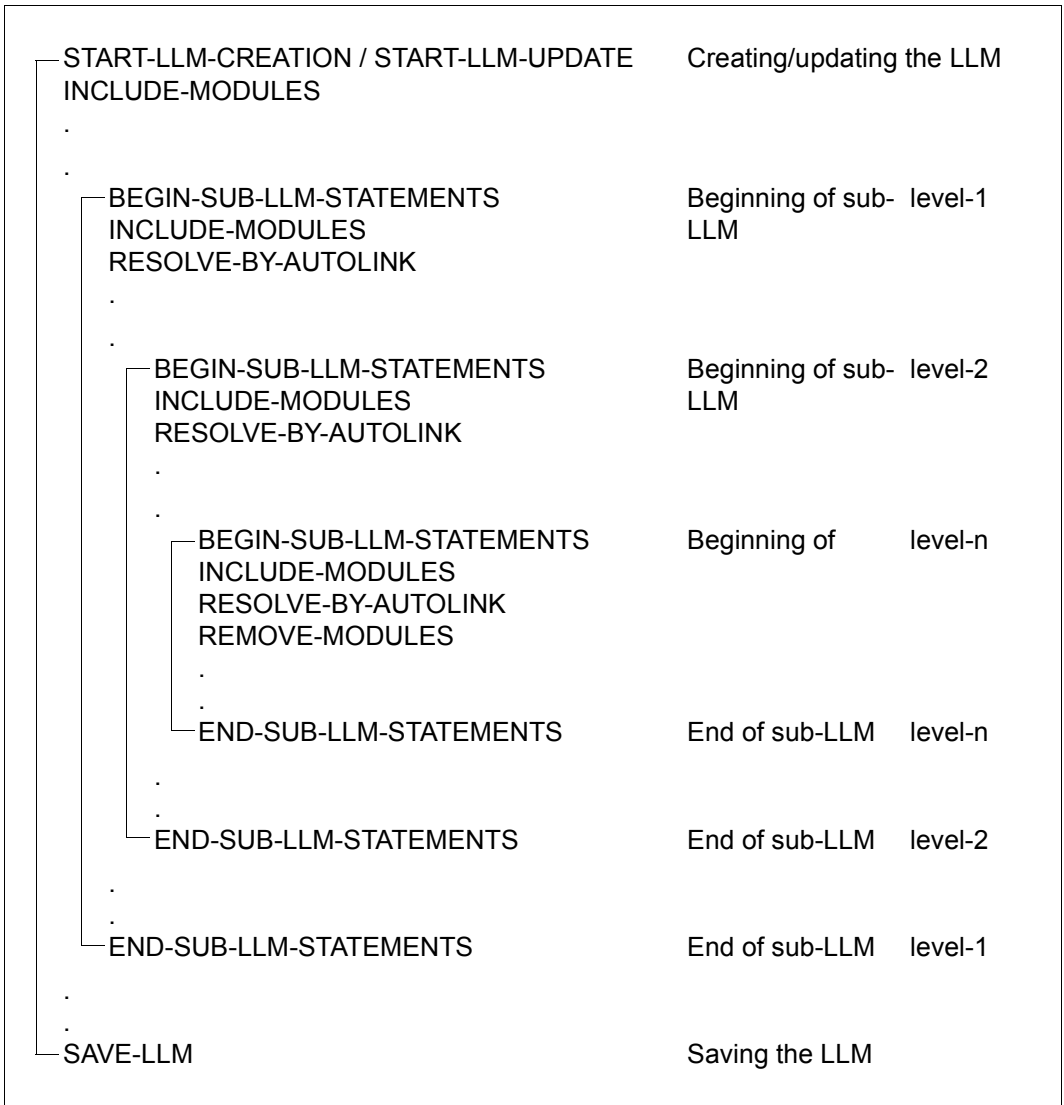


Figure 5: Nesting of sub-LLMs

The following examples explain the effect of the statements BEGIN-SUB-LLM-STATEMENTS and END-SUB-LLM-STATEMENTS.

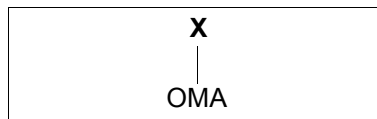
*Example 1*

This example illustrates the simplest use of BEGIN-SUB-LLM-STATEMENTS and END-SUB-LLM-STATEMENTS. Here, two sub-LLMs M1 and M2 are begun and ended *in succession*.

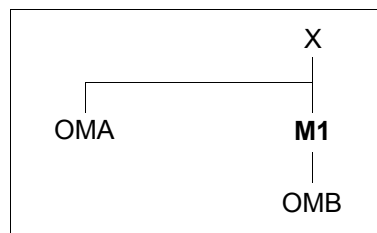
Statements

Current LLM  
(work area)

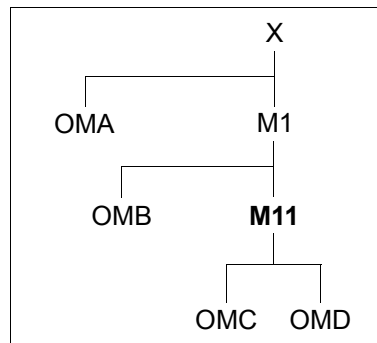
- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=OMA



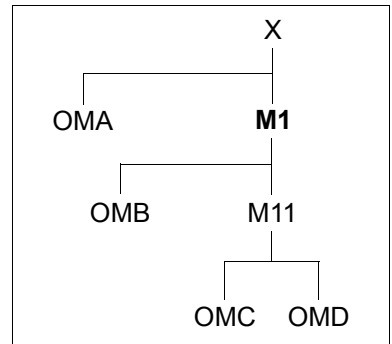
- (3) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=M1
- (4) INCLUDE-MODULES ELEMENT=OMB



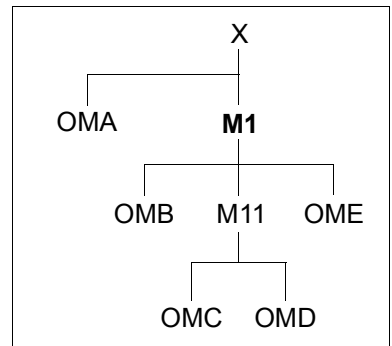
- (5) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=M11
- (6) INCLUDE-MODULES  
ELEMENT=(OMC,OMD)



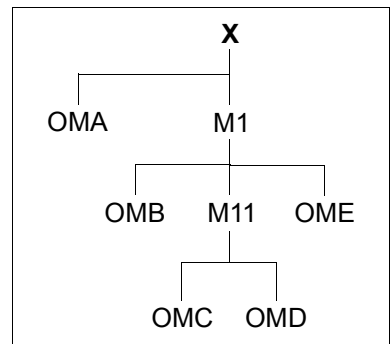
(7) END-SUB-LLM-STATEMENTS



(8) INCLUDE-MODULES ELEMENT=OME

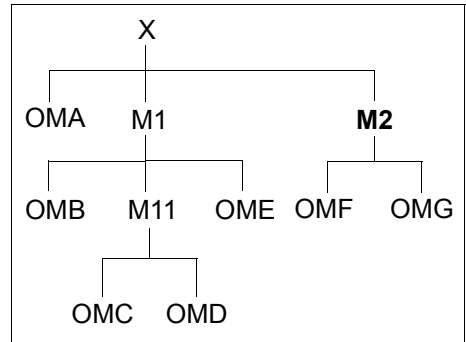


(9) END-SUB-LLM-STATEMENTS



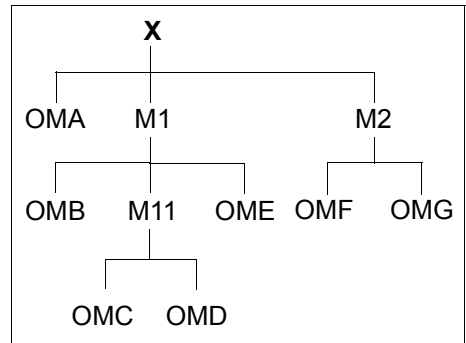
(10) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=M2

(11) INCLUDE-MODULES  
ELEMENT=(OMF,OMG)



(12) END-SUB-LLM-STATEMENTS

(13) SAVE-LLM LIBRARY=LIB1



Explanation:

- (1) An LLM with the internal name X is created in the work area. The root X of the LLM thus becomes the current LLM.
- (2) Object module OMA is included from program library LIB1.
- (3) A sub-LLM with the name M1 is begun at the root. M1 then becomes the current sub-LLM.
- (4) Object module OMB is included in the current sub-LLM M1 from the current program library LIB1.
- (5) A sub-LLM with the name M11 is begun in the current sub-LLM M1. M11 then becomes the current sub-LLM.
- (6) Object modules OMC and OMD are included in the current sub-LLM M11 from the current program library LIB1.
- (7) The current sub-LLM M11 is ended. M1 then becomes the current sub-LLM.

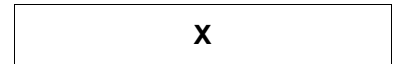
- (8) Object module OME is included in the current sub-LLM M1.
- (9) The current sub-LLM is ended. The root then becomes the current sub-LLM.
- (10) A sub-LLM with the name M2 is begun at the root. M2 then becomes the current sub-LLM.
- (11) Object modules OMF and OMG are included in the current sub-LLM M2.
- (12) The current sub-LLM M2 is ended. The root X then becomes the current LLM.
- (13) The LLM created is saved in program library LIB1. The internal name X from the most recent START-LLM-CREATION statement is taken over as the element name.

*Example 2*

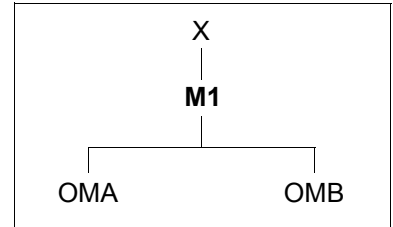
This example illustrates the use of the statements BEGIN-SUB-LLM-STATEMENTS and END-SUB-LLM-STATEMENTS with path names. Here, two sub-LLMs, M1 and M2, are created. After M2 has been created, M1 is supplemented by a sub-LLM, M11.

StatementsCurrent LLM  
(work area)

(1) START-LLM-CREATION  
INTERNAL-NAME=X

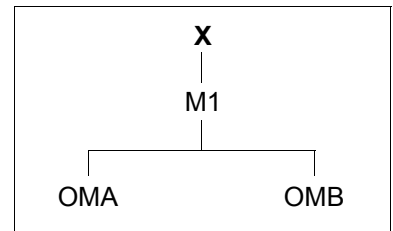


(2) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=M1



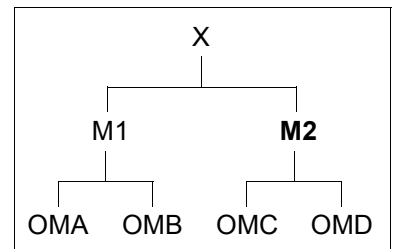
(3) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(OMA,OMB)

(4) END-SUB-LLM-STATEMENTS



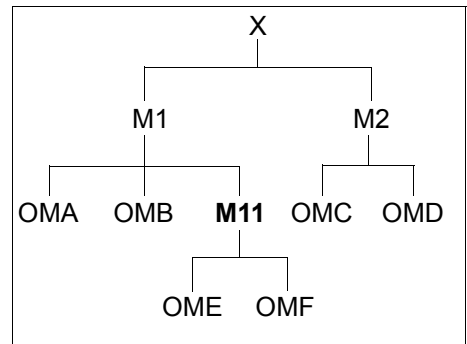
(5) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=M2

(6) INCLUDE-MODULES  
ELEMENT=(OMC,OMD)

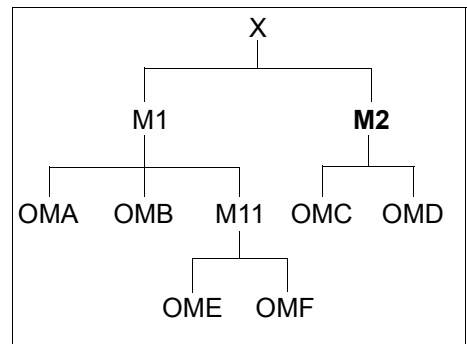


(7) BEGIN-SUB-LLM-STATEMENTS  
 SUB-LLM-NAME=M11  
 PATH-NAME=X.M1

(8) INCLUDE-MODULES  
 ELEMENT=(OME,OMF)

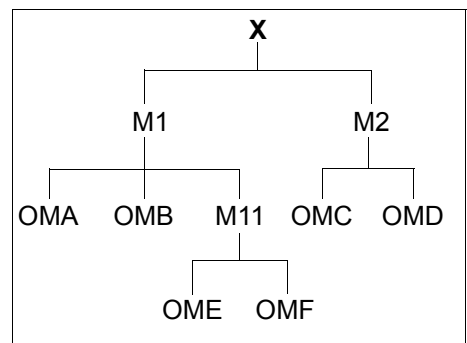


(9) END-SUB-LLM-STATEMENTS



(10) END-SUB-LLM-STATEMENTS

(11) SAVE-LLM LIBRARY=LIB1





Explanation:

- (1) An LLM with the internal name X is created in the work area. The root X of the LLM thus becomes the current LLM.
- (2) A sub-LLM with the name M1 is begun. M1 then becomes the current sub-LLM.
- (3) Object modules OMA and OMB are included in the current sub-LLM M1 from program library LIB1.
- (4) M1 is ended. The root X of the LLM again becomes the current LLM.
- (5) A sub-LLM with the name M2 is begun. M2 becomes the current sub-LLM.
- (6) Object modules OMC and OMD are included in the current sub-LLM M2 from the current program library LIB1.
- (7) A sub-LLM M11 is begun. Path name X.M1 specifies that M11 is to begin in sub-LLM M1. The sub-LLM M11 then becomes the current sub-LLM.
- (8) Object modules OME and OMF are included in the current sub-LLM M11 from the current program library LIB1.
- (9) The sub-LLM M11 is ended. The sub-LLM M2, which was the current sub-LLM prior to the beginning of the sub-LLM M11, again becomes the current sub-LLM.
- (10) The current sub-LLM M2 is ended. The root X again becomes the current sub-LLM.
- (11) The created LLM is saved in program library LIB1.

### 3.3.2 Modifying the logical structure of an LLM

The MODIFY-MODULE-ATTRIBUTES statement can be used to modify the logical structure of an LLM. For this, the path name of a linked module is changed, causing the module to be integrated elsewhere within the LLM.

In addition to the path name, the MODIFY-MODULE-ATTRIBUTES statement can also be used to change the following attributes:

- the logical name of the (sub-)LLM (NEW-NAME)
- declarations for the list for symbolic debugging (LSD) (TEST-SUPPORT)
- masking of all symbols of a module (RUN-TIME-VISIBILITY)
- handling of name conflicts (NAME-COLLISION) if the value of the RUN-TIME-VISIBILITY operand was changed.

The following example shows the effects of the MODIFY-MODULE-ATTRIBUTES statement when modifying the logical structure of an LLM.

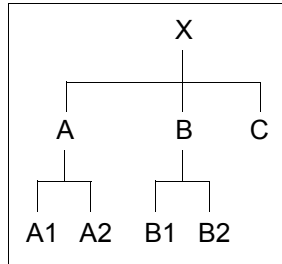
*Example*

Changing the path name of a sub-LLM

Statements

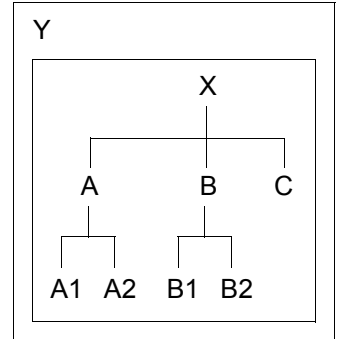
(1) START-LLM-UPDATE  
LIBRARY=LIB1,  
ELEMENT=Y

Current LLM  
(work area)

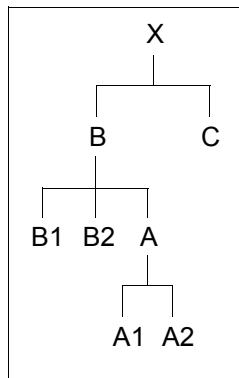


Program library

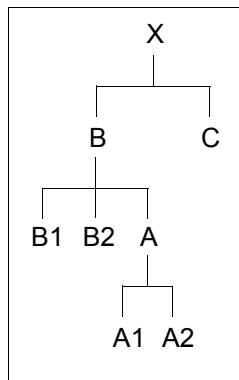
LIB1



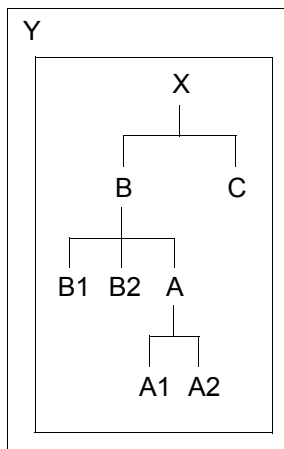
- (2) MODIFY-MODULE-ATTRIBUTES  
 NAME=A,  
 PATH-NAME=X,  
 NEW-PATH-NAME=X.B



- (3) SAVE-LLM  
 LIBRARY=LIB1,  
 ELEMENT=Y



LIB1



Explanation:

- (1) An LLM which is stored as an element with the name Y in program library LIB1 is to be modified. For this, the LLM is loaded into the work area.
- (2) The path name of sub-LLM A is changed from X to X.B, causing the sub-LLM to be integrated elsewhere in the LLM.
- (3) The modified LLM is again stored under element name Y in program library LIB1.

## 3.4 Creating the physical structure of an LLM

The physical structure of an LLM is implemented by means of slices (see [page 18ff](#)). The following describes how to create physical structures of an LLM comprising

- user-defined slices or
- slices by attributes (of CSECTs)

### 3.4.1 User-defined slices

When creating the physical structure of an LLM, the user can define slices that are loaded separately. Initially the loader loads only the **root slice** into main memory. The root slice remains loaded during the entire program run. The user can have the other slices loaded dynamically as soon as they are required for the program run.

Slices may overlay one another, i.e. they occupy the same address space in succession. They have, for example, the same start address and can be loaded, overlaid and reloaded as often as required. The user can request that an Overlay Control Module (OCM) be generated in the root slice of the user program. This OCM controls the LDSLICE macros that are required (see the “BLSSERV Dynamic Binder Loader / Starter” manual [1]) to implement the overlay. If errors occur with LDSLICE (e.g. if an empty slice is to be loaded), the user program is terminated in the 'ABNORMAL' status.

#### Diagrammatic representation of the physical structure

The representation of the desired structure in the form of a diagram helps the user by illustrating the relationships between the individual slices in graphic form (see [figure 6](#)).

Each vertical line in this diagram represents one slice. The uppermost slice in the structure is the **root slice (%ROOT)**. This is loaded at the beginning, while the other slices are loaded only when required.

Each horizontal line is an address level. All slices that start at the same address level have the same load address that immediately follows the preceding slice. They may thus overlay one another. These slices are therefore known as **exclusive slices**. The root slice is never overlaid.

All slices through which a nonbranching line can be drawn lie in a common path. All slices between the root slice and a slice X are referred to as being “higher than X” in the relevant path of the diagram. All slices beneath a slice X are “lower than X”, relative to this slice. The root slice is thus higher than all other slices. All exclusive slices are lower than the root slices.

Each slice is assigned a **level number** that is determined by the number of higher slices in the relevant path. The root slice has the level zero. A new **region** begins at an address level that is situated such that overlaying by preceding slices is excluded.

The user can have an overview of the physical structure of the current LLM output at any time by means of the SHOW-MAP statement.

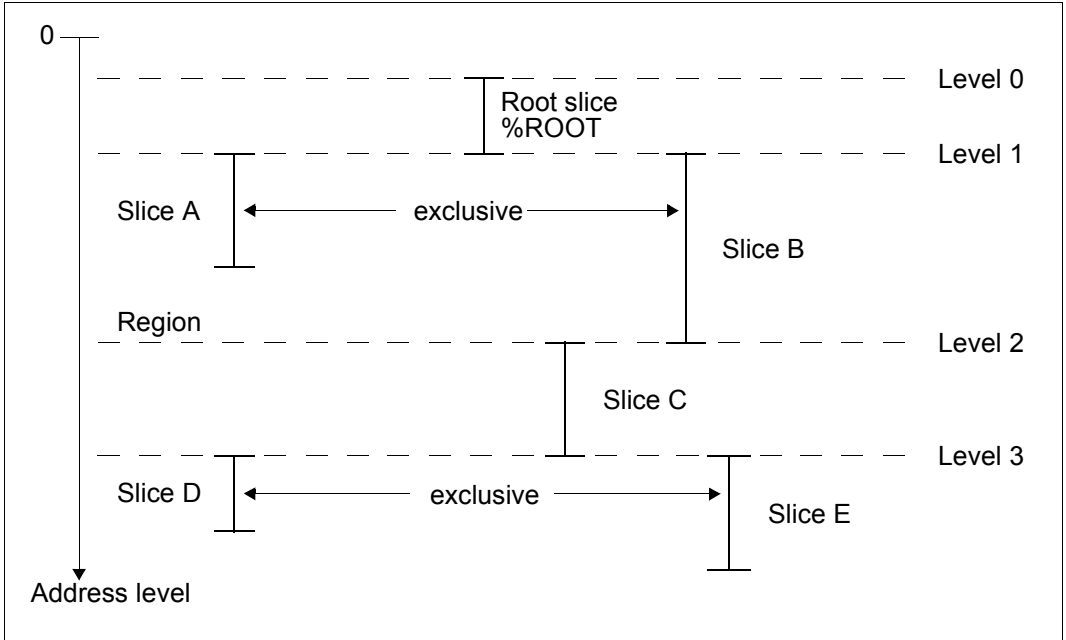


Figure 6: Example of diagrammatic representation of the physical structure

### Defining the physical structure

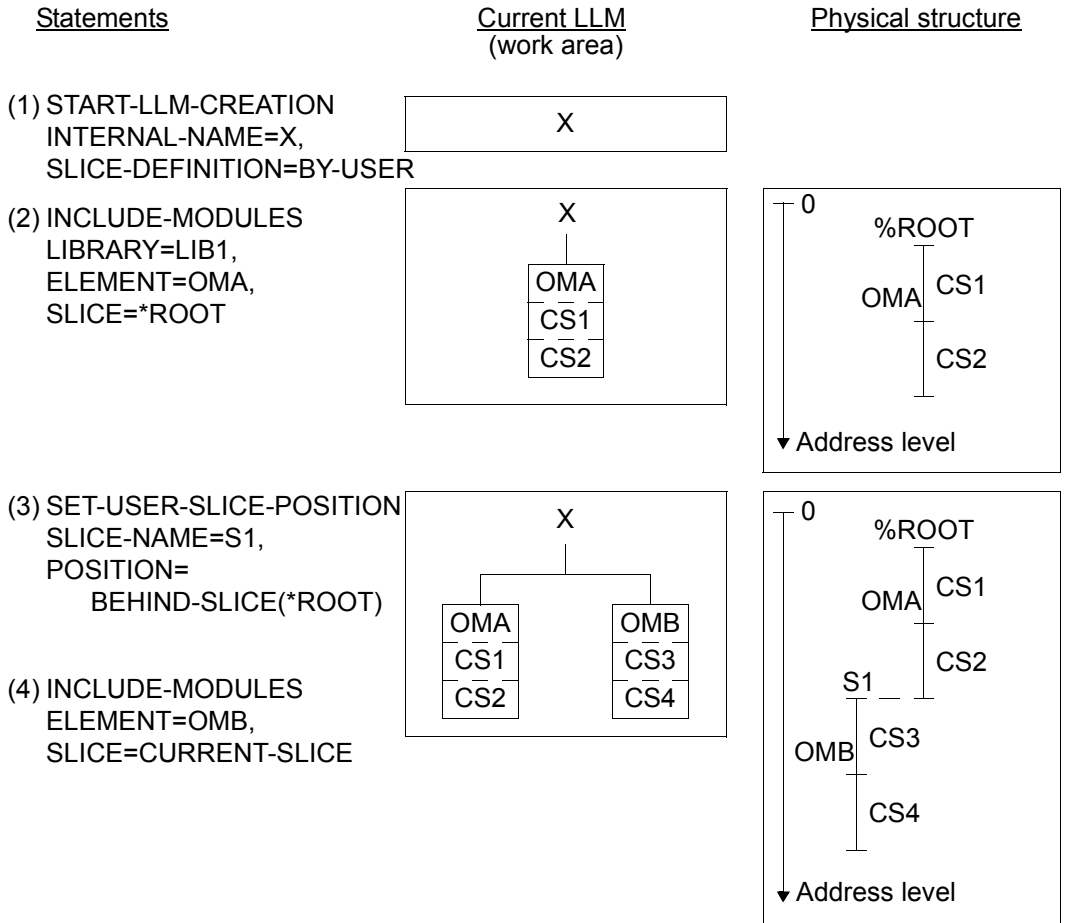
The user defines the physical structure using SET-USER-SLICE-POSITION statements. The slice specified in a SET-USER-SLICE-POSITION statement becomes the **current slice**. This is the slice in which modules are included or replaced if no specifications for the slice are made in the INCLUDE-MODULES or REPLACE-MODULES statements (SLICE operand).

If no slice has yet been defined with a SET-USER-SLICE-POSITION statement, the root slice (%ROOT) is assumed as the current slice. Modules selected by autolink are only linked into the root slice.

If the user wishes to use SET-USER-SLICE-POSITION statements to define the physical structure, this must be declared in the START-LLM-CREATION statement on creating the LLM (SLICE-DEFINITION=BY-USER operand).

*Example 1*

Creating a physical structure comprising the root slice and two exclusive slices

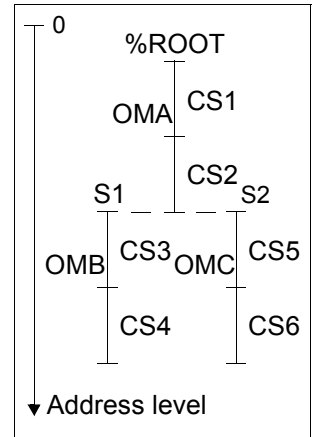
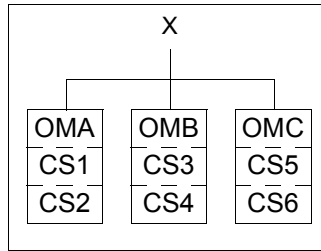


(5) SET-USER-SLICE-POSITION  
ON

SLICE-NAME=S2,  
POSITION=

(6) INCLUDE-MODULES

ELEMENT=OMC,  
SLICE=S2



(7) SAVE-LLM LIBRARY=LIB1

Explanation:

- (1) An LLM with the internal name X is created in the work area. The SLICE-DEFINITION operand declares that the physical structure of the LLM is defined by the user with SET-USER-SLICE-POSITION statements. The root slice %ROOT is defined.
- (2) Object module OMA is included in the current LLM in the work area from program library LIB1. OMA contains the two CSECTs CS1 and CS2. The SLICE=\*ROOT operand defines that CS1 and CS2 constitute the root slice %ROOT.
- (3) The slice S1 is defined. It immediately follows the root slice %ROOT. S1 becomes the current slice.
- (4) Object module OMB is included in the current LLM from program library LIB1. OMB contains the two CSECTs CS3 and CS4. The CSECTs constitute the slice S1 (current slice).
- (5) The slice S2 is defined. It immediately follows the root slice %ROOT. S2 becomes the current slice. The slices S2 and S1 are mutually exclusive, i.e. S2 can overlay S1.
- (6) Object module OMC is included in the current LLM from program library LIB1. OMC contains the two CSECTs CS5 and CS6. The SLICE=S2 operand defines that CS5 and CS6 constitute the slice S2.
- (7) The current LLM is saved in program library LIB1.

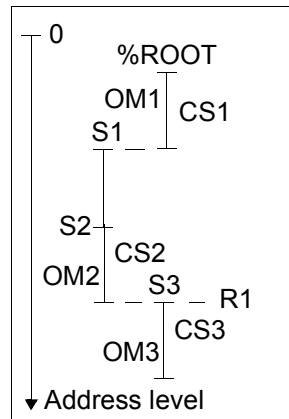
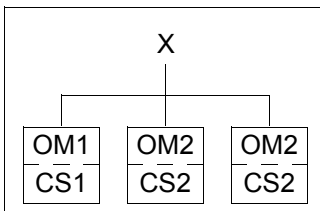
*Example 2*

Creating a physical structure comprising the root slice, two exclusive slices and a new region

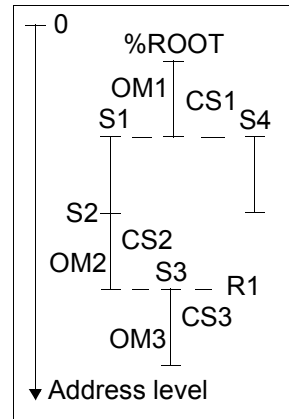
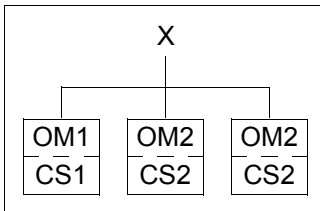
<u>Statements</u>	<u>Current LLM</u> (work area)	<u>Physical structure</u>
(1) START-LLM-CREATION INTERNAL-NAME=X, SLICE-DEFINITION=BY-USER		
(2) INCLUDE-MODULES LIBRARY=LIB1, ELEMENT=OM1, SLICE=*ROOT		
(3) SET-USER-SLICE-POSITION SLICE-NAME=S1, POSITION= BEHIND-SLICE(*ROOT)		
(4) SET-USER-SLICE-POSITION SLICE-NAME=S2, POSITION= BEHIND-SLICE(*CURRENT)		
(5) INCLUDE-MODULES ELEMENT=OM2, SLICE=CURRENT-SLICE		



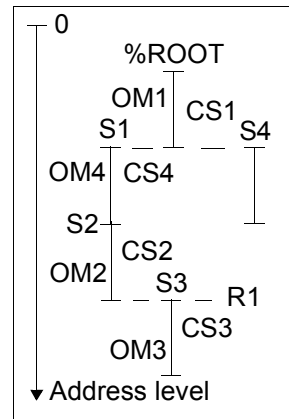
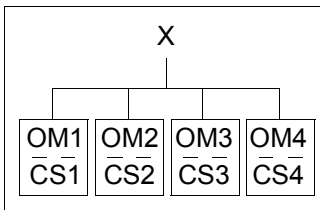
- (6) SET-USER-SLICE-POSITION  
SLICE-NAME=S3,  
POSITION=BEGIN-REGION  
(REGION=R1,
- (7) INCLUDE-MODULES  
ELEMENT=OM3,  
SLICE=S3



- (8) SET-USER-SLICE-POSITION  
SLICE-NAME=S4,  
POSITION=  
BEHIND-SLICE(\*ROOT)



- (9) SET-USER-SLICE-POSITION  
SLICE-NAME=S1,
- (10) INCLUDE-MODULES  
ELEMENT=OM4,  
SLICE=S1



- (11) SAVE-LLM LIBRARY=LIB1

Explanation:

- (1) An LLM with the internal name X is created in the work area. The SLICE-DEFINITION operand declares that the physical structure of the LLM is defined by the user with SET-USER-SLICE-POSITION statements. The root slice %ROOT is defined.
- (2) Object module OM1 is included in the current LLM in the work area from program library LIB1. OM1 contains the CSECT CS1. The SLICE=\*ROOT operand defines that CS1 constitutes the root slice %ROOT.
- (3) The slice S1 is defined. It immediately follows the root slice. S1 becomes the current slice.
- (4) The slice S2 is defined. It immediately follows the current slice (\*CURRENT operand). S2 becomes the current slice.
- (5) Object module OM2 is included in the current LLM in the work area from program library LIB1. OM2 contains the CSECT CS2. The CSECT CS2 constitutes the slice S2 (current slice).
- (6) The slice S3 is defined. It begins at a new region R1. Its start address lies at the end of S1 since the slices of region R1 may not overlay the higher slices. S3 becomes the current slice.
- (7) Object module OM3 is included in the current LLM in the work area from program library LIB1. OM3 contains the CSECT CS3. The SLICE=S3 operand defines that CS3 constitutes the slice S3.
- (8) The slice S4 is defined. It immediately follows the root slice %ROOT. S4 becomes the current slice. Slice S4 is exclusive with respect to slices S1 and S2, i.e. S4 can overlay S1 and S2.
- (9) The existing slice S1 is to be updated (MODE=UPDATE operand).
- (10) Object module OM4 is included in the current LLM in the work area from program library LIB1. OM4 contains the CSECT CS4. The SLICE=S1 operand defines that CS4 constitutes the slice S1.
- (11) The current LLM is saved in program library LIB1.

### 3.4.2 Slices by attributes

#### Procedure for forming slices

For an LLM whose slices are to be formed on the basis of attributes of CSECTs, BINDER automatically forms slices on the basis of the following attributes:

##### *READ-ONLY*

- Read access (READ-ONLY=YES)  
The CSECT can only be read. This attribute protects the CSECT in main memory against overwriting.
- Read and write access (READ/WRITE) (READ-ONLY=NO)  
The CSECT can be read and overwritten.

##### *RESIDENT*

- Main memory resident (RESIDENT=YES)  
The CSECT is loaded into class 3 memory and held resident there.
- Pageable (PAGEABLE) (RESIDENT=NO)  
The CSECT is pageable.  
  
This attribute is relevant *only* for shared code of the system.

##### *PUBLIC*

- Shareable (PUBLIC=YES)  
The CSECT contains data and instructions available for shared use (see [page 76](#)).
- Private (PRIVATE, i.e. PUBLIC=NO)  
The CSECT contains data and instructions available for private use only.

##### *RMODE*

- Residence mode (RMODE=ANY)  
The CSECT can be loaded below 16 Mb and above 16 Mb.
- Residence mode (RMODE=24)  
The CSECT can only be loaded below 16 Mb.

By combining all these attributes, up to 16 different slices can be formed. The attributes on which slices are based are defined by the user in the START-LLM-CREATION statement (BY-ATTRIBUTES operand).

If the user specifies, for example, that slices are to be formed on the basis of the attribute READ-ONLY, BINDER generates the following two slices:

1. one slice with all CSECTs READ-ONLY
2. one slice with all CSECTs READ/WRITE

All other attributes are not used for forming slices since the default values NO apply in the BY-ATTRIBUTES operand.

If no CSECT having the attribute READ-ONLY is present, no slice is generated. BINDER does not therefore generate an empty slice. All slices are loaded into main memory as an entire unit. In the above example, the READ-ONLY slices and the READ/WRITE slices are loaded into class 6 memory.

### Slices with the attribute PUBLIC

Slices with the attribute PUBLIC=YES must be programmed as reentrant, since this is a prerequisite for their use as shared code (see the “BLSSERV Dynamic Binder Loader / Starter” manual [1]). All public slices of an LLM can be loaded only together as shared code and form the PUBLIC part of this LLM. There are three possible ways of loading the PUBLIC part of an LLM as shared code:

1. The user can load the public slices with DBL macro ASHARE in a common memory pool in class 6 memory. The public slices can be unloaded with DBL macro DSHARE.
2. With DSSM, the public slices can be loaded as an unprivileged subsystem (see the “Introductory Guide to Systems Support” [9]) in class 3 or class 4 memory. They can be unloaded only via the DSSM.

When such an LLM is called with the LOAD-EXECUTABLE-PROGRAM or START-EXECUTABLE-PROGRAM (or LOAD-PROGRAM/START-PROGRAM) command or with DBL macro BIND, the private slices are loaded into the task-local class 6 memory. In order to resolve the external references, the system searches the entire shared code for the public slices belonging to the LLM. This search can be prohibited with the SHARE[-SCOPE] operand in the program call (START-EXECUTABLE-PROGRAM, LOAD-EXECUTABLE-PROGRAM, START-PROGRAM, LOAD-PROGRAM, BIND macro) (see the “BLSSERV Dynamic Binder Loader / Starter” manual [1]). External references in the private slices are resolved by the public slices, but external references in the public slices are *not* resolved by the private slices. If the PUBLIC part of the LLM is not found in the shared code, loading of the PUBLIC part into the task-local class 6 memory is initiated.

## Slice name

The slice names are produced by BINDER. They are logged in the lists output by BINDER (see [page 141ff](#)).

The slice name consists of 4 subnames (one subname for each attribute) and has the following structure:

PUa-ROb-RTc-RMd

where:

PUa	Subname for the attribute PUBLIC
a	Identifier for the type of slice
a=U	Attribute not used for forming the slice (UNDEFINED)
a=Y	Slice is sharerable (PUBLIC=YES)
a=N	Slice is not sharerable (PUBLIC=NO)
ROb	Subname for the attribute READ-ONLY
b	Identifier for the type of slice
b=U	Attribute not used for forming the slice (UNDEFINED)
b=Y	Slice with read access (READ-ONLY=YES)
b=N	Slice with read and write access (READ-ONLY=NO)
RTc	Subname for the attribute RESIDENT
c	Identifier for the type of slice
c=U	Attribute not used for forming the slice (UNDEFINED)
c=Y	Slice is main memory resident (RESIDENT=YES)
c=N	Slice is not main memory resident (RESIDENT=NO)
RMd	Subname for the attribute RESIDENCY-MODE
d	Identifier for the type of slice
d=U	Attribute not used for forming the slice (UNDEFINED)
d=A	RMODE=ANY
d=4	RMODE=24

### *Example*

If only the attribute READ-ONLY is to be used for forming slices, two slices with the following slice names are formed:

1. PUU-ROY-RTU-RMU Slice with read access
2. PUU-ROn-RTU-RMU Slice with read and write access

### 3.4.3 Modifying the type of physical structure

The type of physical structure can be modified later with the MODIFY-LLM-ATTRIBUTES statement (SLICE-DEFINITION operand). The type of physical structure may be modified as follows:

1. LLM with slices by attributes → LLM with single slice
2. LLM with single slice → LLM slices by attributes
3. LLM with slices by attributes → LLM with slices by *other* attributes
4. LLM with user-defined slices → LLM with user-defined slices and modified values for AUTOMATIC-CONTROL and EXCLUSIVE-SLICE-CALL.

### 3.4.4 Connection between private and public slices

The connection between private and public slices can be controlled by means of the CONNECTION-MODE operand. This operand is subordinate to the FOR-BS2000-VERSIONS operand in statements MODIFY-STD-DEFAULTS and SAVE-LLM.

The CONNECTION-MODE operand is applied only if the LLMs have been divided into slices in accordance with the PUBLIC attribute of the CSECTs.

The connection between private and public slices can be made in two ways:

1. by resolution (CONNECTION-MODE=\*BY-RESOLUTION):  
While loading, the DBL sets up connections between the private part and all symbols (name and type) in the PUBLIC part to which relocation information from the private part refers.
2. by relocation (CONNECTION-MODE=\*BY-RELOCATION):  
While loading, the DBL sets up a single connection between the private part and the PUBLIC part. If a subsystem ENTRY is specified, the connection refers to it.

#### *Restriction*

With LLM format 1, a connection between the private part and the PUBLIC part is set up only when the public slice contains at least one definition to which an external reference in the private part refers.

With CONNECTION-MODE=\*BY-RELOCATION the performance of DBL when setting up the connection is better than with CONNECTION-MODE=\*BY-RESOLUTION. If, however, you are using the indirect linkage mechanism (see the “BLSSERV Dynamic Binder Loader/Starter” manual [1]) to connect to modules in the public slice, you should use CONNECTION-MODE=\*BY-RESOLUTION.

*Note*

If you have defined subsystem entries with the statement MODIFY-LLM-ATTRIBUTES, the information on this is output to the stored LLM, regardless of its format. With CONNECTION-MODE=\*BY-RESOLUTION, specifications of subsystem entries have no effect.

## 3.5 Resolving external references

External references in the modules that are included in the LLM refer to a control section (CSECT), an entry point (ENTRY) or a COMMON in a different module. BINDER attempts to resolve all external references immediately, i.e. it searches within the created LLM for a CSECT, an ENTRY or a COMMON with the same name and enters the address it has found in the external reference.

To satisfy external references that are still unresolved the user can call the BINDER **autolink function** using the RESOLVE-BY-LINK statement. BINDER searches the libraries specified in RESOLVE-BY-AUTOLINK for CSECTs, ENTRYs and COMMONs in modules that satisfy the unresolved external references. When a module that resolves an external reference is found, it is included in the current LLM.

The user can specify that input libraries assigned through the file link name BLSLIBnn (00≤nn≤99) are to be searched.

The libraries are searched in ascending order of values “nn” for this file link name. They must be assigned prior to the BINDER run. These file link names are *not* released after a BINDER run.

The autolink function primarily saves users of higher-level programming languages having to repeatedly enter INCLUDE-MODULES statements in order to include the often fairly numerous modules required for the runtime system.

Weak external references (WXTRNs) *cannot* be resolved by autolink. The names of unresolved WXTRNs are logged by BINDER after execution of the autolink function in the list of unresolved weak external references (see [page 141ff](#)). External references that are not referenced in the program cannot be resolved by autolink either.

### *Note*

The time required by autolink depends essentially on how many libraries need to be searched before suitable modules are found. In the worst case, all specified libraries have to be searched.

A list of unresolved external references is output provided the user does not suppress it with the SHOW-MAP or MODIFY-MAP-DEFAULTS statement (see [page 141ff](#)).



Depending on the mode under which BINDER is operating, the user can take the following courses of action:

#### *Interactive mode*

The user can enter further INCLUDE-MODULES or RESOLVE-BY-AUTOLINK statements in order to resolve any outstanding unresolved external references. If the user does not wish to do this, BINDER will handle the unresolved external references as defined in the SET-EXTERN-RESOLUTION statement (see [page 96ff](#)).

#### *Batch mode*

Unresolved external references are handled as defined in the SET-EXTERN-RESOLUTION statement (see [page 96ff](#)).

### 3.5.1 Rules for resolving external references

Initially BINDER attempts to resolve external references with names of CSECTs, ENTRYs and COMMONs from modules that it has included in the current LLM. However, an external reference is never resolved with a name from a module included in the FORBIDDEN-SCOPE of the module which contains the reference.

The following rules are applicable here:

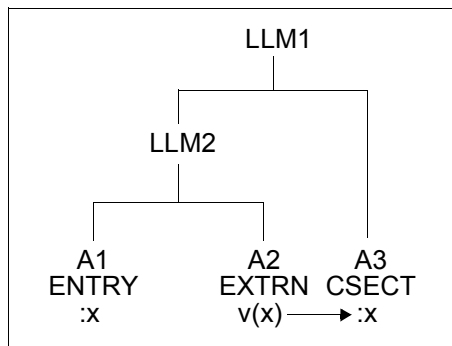
- **Rule 1**  
CSECTs, ENTRYs and COMMONs in the HIGH-PRIORITY-SCOPE of the module which contains the external reference have priority over all other program definitions. CSECTs, ENTRYs and COMMONs in the LOW-PRIORITY-SCOPE of this module are only applied after all other program definitions.
- **Rule 2**  
When a module is included in an LLM, the names of CSECTs, ENTRYs and COMMONs within the same sub-LLM have priority over the names of CSECTs, ENTRYs and COMMONs in other branches. During searching, the search commences with the sub-LLM on the *highest* level that contains references and program definitions.
- **Rule 3**  
A CSECT has priority over an ENTRY, and an ENTRY has priority over a COMMON.
- **Rule 4**  
The external references are processed *in succession*. The OMs and sub-LLMs of the current LLM are processed here in the order in which they are arranged in the logical structure (in the LLM structure tree, from left to right).

LLMs with user-defined slices are an exception to these rules. For these LLMs, the following search sequence has priority over the three standard rules:

1. Search for definitions in the slice which also contains the external reference.
2. Search for definitions in the slices with the same path and same region as the slice containing the external reference.
3. Search for definitions in slices following the slice containing the external reference.
4. Search for definitions in slices located in other regions.
5. Search for definitions in competing slices if START-LLM-CREATION ... EXCLUSIVE-SLICE-CALL=YES was specified.

*Example 1 (rule 1)*

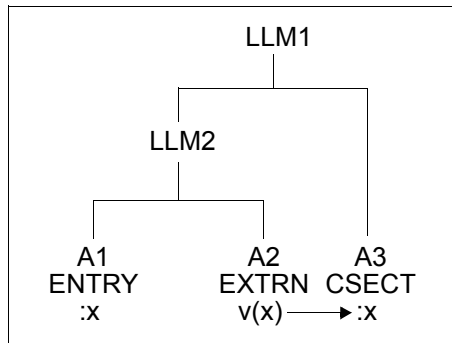
Current LLM



Requirement:

A HIGH-PRIORITY-SCOPE containing object module A3 must have been defined for object module A2.

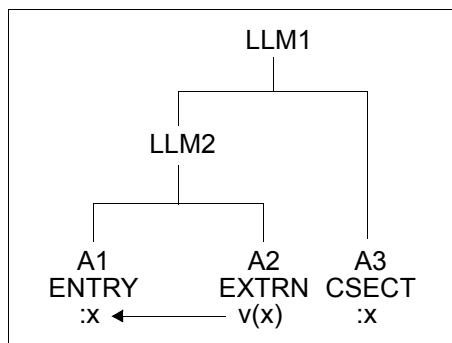
To resolve the EXTRN  $v(x)$  in sub-LLM A2, BINDER searches the HIGH-PRIORITY-SCOPE of A2 first; in other words, A3 and not A1, as would be the case without RESOLUTION-SCOPE. ENTRY  $x$  in A3 resolves EXTRN  $v(x)$ .

*Example 2 (rule 1)*Current LLM

## Requirement:

A LOW-PRIORITY-SCOPE containing object module A1 must have been defined for sub-LLM LLM2.

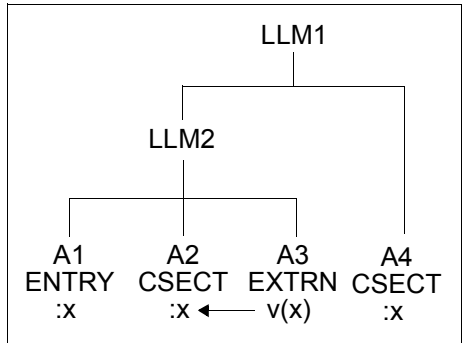
Object module A2 inherits the LOW-PRIORITY-SCOPE from the parent node, i.e. from sub-LLM LLM2. BINDER starts its search for the unresolved EXTRN v(x) in object module A2 in modules which are not contained in the LOW-PRIORITY-SCOPE of A2. EXTRN v(x) is therefore resolved with ENTRY x in A3.

*Example 3 (rule 2)*Current LLM

Object module A2 with the EXTRN v(x) is included in sub-LLM LLM2. The EXTRN v(x) in object module A2 is resolved by ENTRY x in object module A1 because A2 and A1 lie in the same sub-LLM LLM2. The CSECT x in object module A3 lies in a different branch and is ignored.

Example 4 (rules 2 and 3)

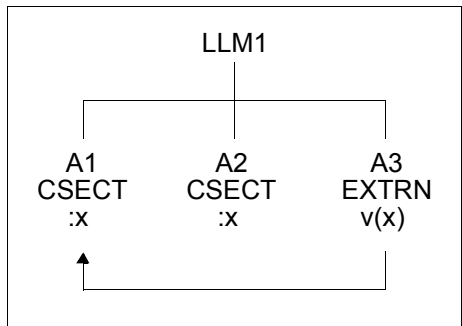
Current LLM



Object module A3 with the EXTRN v(x) is included in sub-LLM LLM2. The EXTRN v(x) in object module A3 is resolved by the CSECT x in object module A2 because A3 and A2 lie in the same sub-LLM LLM2. Within LLM2 the CSECT x in A2 has priority over the ENTRY x in A1.

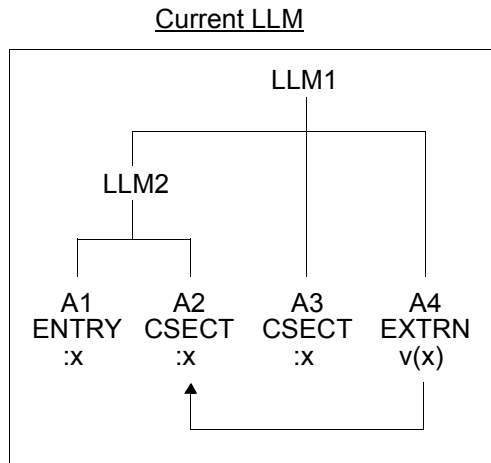
Example 5 (rule 4)

Current LLM



Object module A3 with the EXTRN v(x) is included in the LLM1. The EXTRN v(x) in object module A3 is resolved by the CSECT x in object module A1 because the LLM structure tree is searched from left to right.

Example 6 (rules 2 and 3)



Object module A4 with the EXTRN  $v(x)$  is included in the LLM1. The EXTRN  $v(x)$  in object module A4 is resolved by the CSECT  $x$  in object module A2 in accordance with the following rules:

- Searching commences with the sub-LLM on the highest level. This is the sub-LLM LLM2 (rule 1).
- Within the sub-LLM LLM2, the CSECT  $x$  in object module A2 has a higher priority than the ENTRY  $x$  in object module A1 (rule 2).

### 3.5.2 Autolink function

The autolink function of BINDER enables the automatic inclusion of modules. In order to resolve external references, BINDER searches for CSECTs and ENTRYs in those modules and libraries specified in the RESOLVE-BY-AUTOLINK statement. The following rules are applicable:

- **Rule 1**

The external references are processed *in succession*. The OMs and sub-LLMs of the current LLM are processed in the order in which they are arranged in the logical structure (in the LLM structure tree, from left to right).

Within an OM the external references are processed in accordance with their order in the OM.

With the modules of a program library, all elements are searched in accordance with the specified type (TYPE operand).

- **Rule 2**  
When a module is included in order to resolve an external reference, BINDER also attempts to resolve further unresolved external references within the whole LLM by using CSECTs, ENTRYs and COMMONs of this module.
- **Rule 3**  
If *more than one* library is specified in a RESOLVE-BY-AUTOLINK statement, the libraries are searched in the order in which they were specified in the statement. For each individual library the modules are searched in accordance with *rule 1*.
- **Rule 4**  
A scope can be specified in the RESOLVE-BY-AUTOLINK statement (SCOPE operand). This scope defines the portion of the LLM structure tree within which external references are resolved. External references outside this scope are not resolved.
- **Rule 5**  
A path name can be specified in the RESOLVE-BY-AUTOLINK statement (PATH-NAME operand). This defines in which sub-LLM of the current LLM modules are included.
- **Rule 6**  
New external references that occur during the inclusion of modules are entered in the list of *unresolved* external references, assuming they are defined in the specified scope (SCOPE operand) and in the valid path (PATH-NAME operand).

For the modules included with RESOLVE-BY-AUTOLINK, the user can specify whether or not they are to be regarded as runtime modules (RUN-TIME-VISIBILITY operand). If the included module is to be a runtime module, all symbols are masked when this module is stored. This can be used to avoid name conflicts when the LLM is loaded by DBL. This masking of the symbols is, however, canceled if the module is read in again (e.g. with INCLUDE-MODULES or START-LLM-UPDATE).

*Note*

The autolink function is not executed if the LLMs contain no relocation information, no logical structure information or no External Symbols Vector, or if LLMs containing user-defined slices are to be included. The autolink function is aborted in this case. The user is responsible for ensuring that libraries contain no such modules.

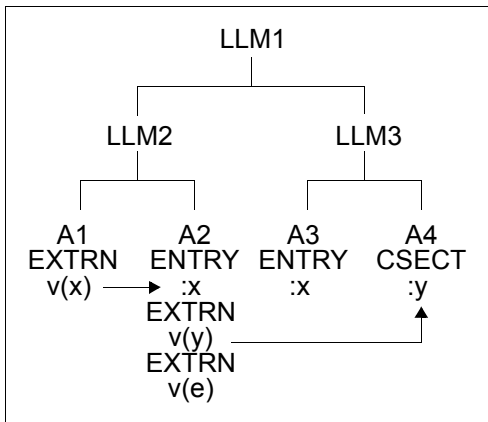
*Example 1*

Stage 1:

The EXTRN v(x) in object module A1 is resolved by the ENTRY x in object module A2 because A1 and A2 lie in the same sub-LLM. The ENTRY x in object module A3 lies in a different branch and is ignored.

The EXTRN v(y) in object module A2 is resolved by the CSECT y in object module A4.

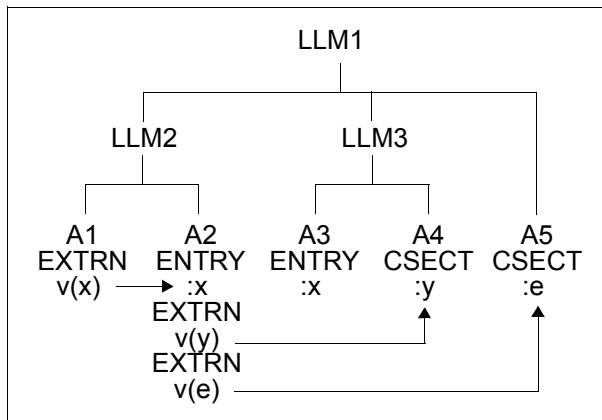
Current LLM



Stage 2:

The still unresolved EXTRN v(e) in object module A2 is resolved by the CSECT e in object module A5 that is included by autolink.

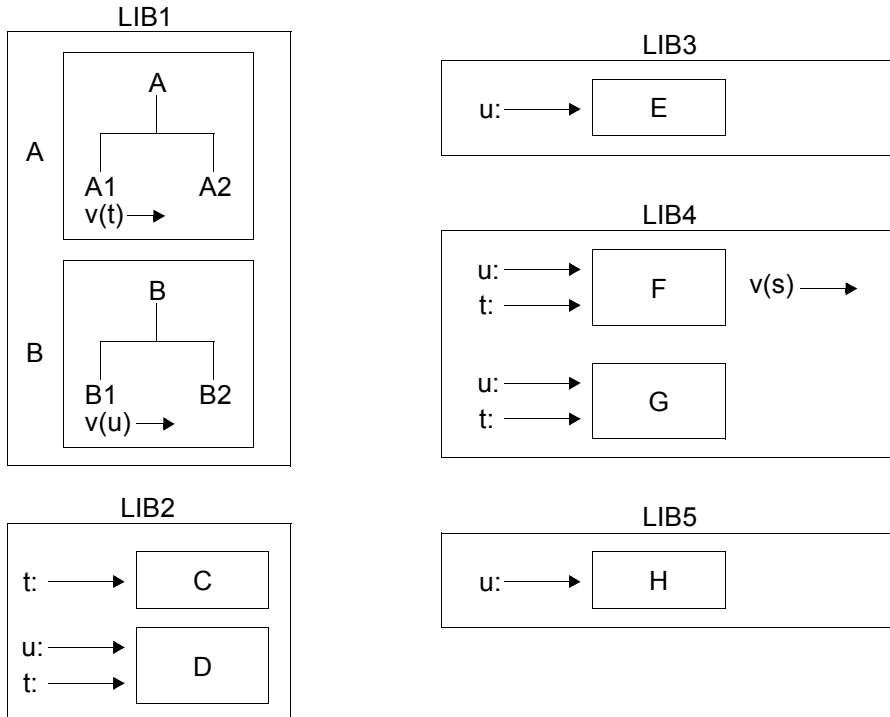
Current LLM



Example 2

The following program libraries (see below) are assumed:

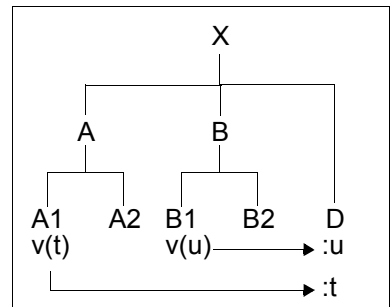
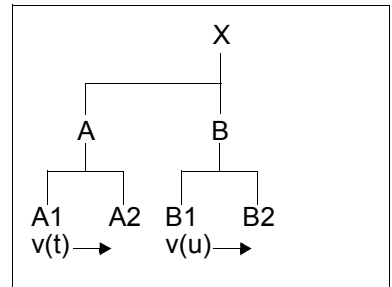
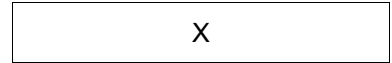
- A program library LIB1.  
This contains the LLM A with an unresolved EXTRN v(t) and the LLM B with an unresolved EXTRN v(u).
- A program library LIB2.  
This contains the OM C with the ENTRY t and the OM D with two ENTRYs u and t.
- A program library LIB3.  
This contains the OM E with the ENTRY u.
- A program library LIB4.  
This contains the following modules:
  - the OM F with the unresolved EXTRN v(s) and two ENTRYs u and t,
  - the OM G with two ENTRYs s and t.
- A program library LIB5.  
This contains the OM H with the ENTRY u.





*Case 1 (rules 1 and 2)*Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L
- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB2,TYPE=(L,R),  
SYMBOL-NAME=U

Current LLM  
(work area)

## Explanation:

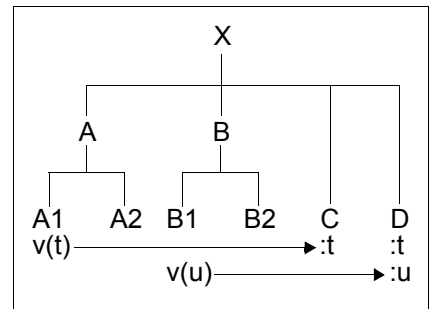
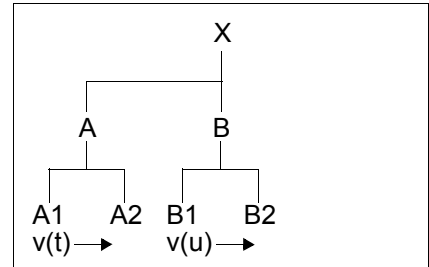
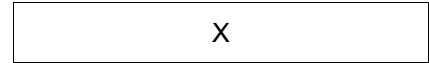
- (1) An LLM with the internal name X is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1.
- (3) Program library LIB2 is searched for the unresolved EXTRN v(u). The OM D resolves the EXTRN v(u). BINDER includes the OM D at the root of the current LLM and attempts to resolve the unresolved EXTRN v(t) with an ENTRY in the OM D. The ENTRY t in the OM D resolves the EXTRN v(t).

Case 2 (rule 1)

Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
  
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L
  
- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB2,TYPE=(L,R)

Current LLM  
(work area)



Explanation:

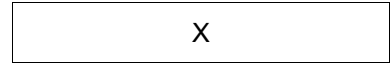
- (1) An LLM with the internal name X is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1.
- (3) Program library LIB2 is searched for *all* unresolved EXTRNs (default value SYMBOL-NAME=\*ALL). The OM C resolves the EXTRN v(t) and the OM D resolves the EXTRN v(u). BINDER includes the OMs C and D at the root of the current LLM.

Case 3 (rules 3, 4 and 5)

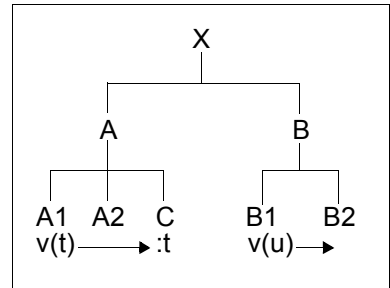
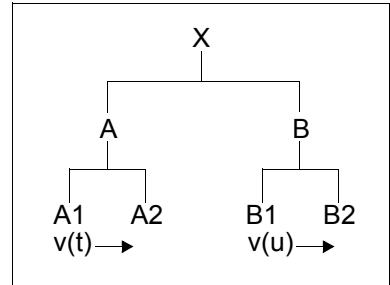
Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L

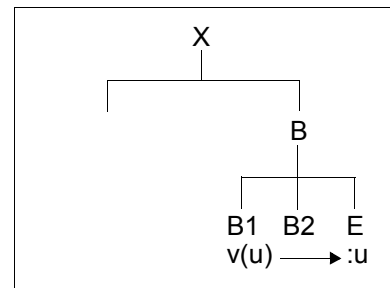
Current LLM  
(work area)



- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB2,TYPE=(L,R),  
SYMBOL-NAME=T,  
PATH-NAME=X.A



- (4) RESOLVE-BY-AUTOLINK  
LIBRARY=(LIB3,LIB4,LIB5),  
TYPE=(L,R),  
SCOPE=EXPLICIT(WITHIN-SUB-LLM=B),  
PATH-NAME=X.B



Explanation:

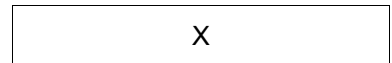
- (1) An LLM with the internal name X is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1.
- (3) Program library LIB2 is searched for the unresolved EXTRN v(t). The first associated ENTRY t is found in OM C. BINDER includes the OM C in the sub-LLM A (path name X.A).
- (4) Program libraries LIB3, LIB4, LIB5 are searched here in the order LIB3 → LIB4 → LIB5 for all as yet unresolved EXTRNs (default value SYMBOL-NAME=\*ALL). Only the unresolved EXTRNs within the sub-LLM B are to be encompassed by the search (SCOPE operand). The first associated ENTRY u is found in LIB3 in the OM E. BINDER includes the OM E in the sub-LLM B (path name X.B).

Case 4 (rules 3, 4 and 5)

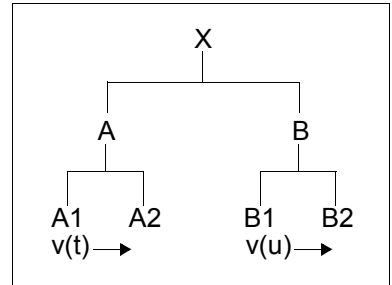
Statements

Current LLM  
(work area)

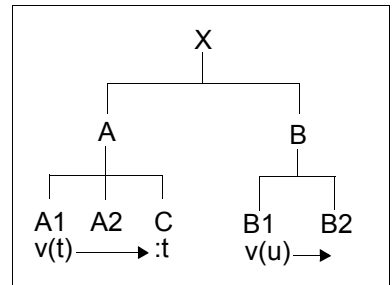
- (1) START-LLM-CREATION  
INTERNAL-NAME=X



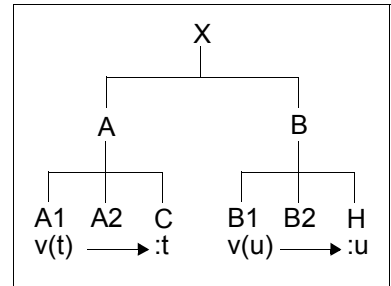
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L



- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB2,TYPE=(L,R),  
SYMBOL-NAME=T,  
PATH-NAME=X.A



- (4) RESOLVE-BY-AUTOLINK  
 LIBRARY=(LIB3,LIB4,LIB5),  
 TYPE=(L,R),  
 SCOPE=EXPLICIT(WITHIN-SUB-LLM=B),  
 PATH-NAME=X.B



Explanation:

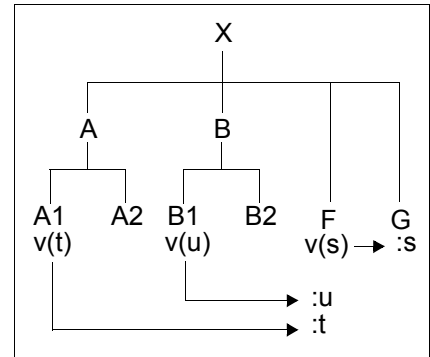
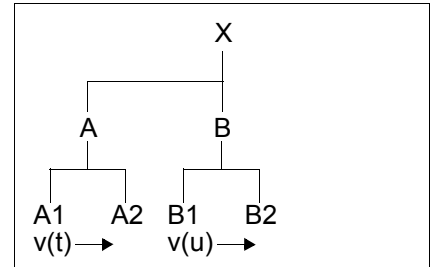
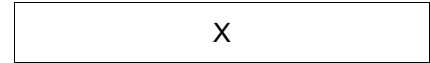
(1), (2), (3) As in case 3.

- (4) Program libraries LIB3, LIB4, LIB5 are searched here in the order LIB5 → LIB4 → LIB3 for all as yet unresolved EXTRNs (default value SYMBOL-NAME=\*ALL). Only the unresolved EXTRNs within the sub-LLM B are to be encompassed by the search (SCOPE operand). The first associated ENTRY u is found in LIB5 in the OM H. BINDER includes the OM H in the sub-LLM B (path name X.B).

## Case 5 (rule 6)

Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L
- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB4,TYPE=(L,R)

Current LLM  
(work area)

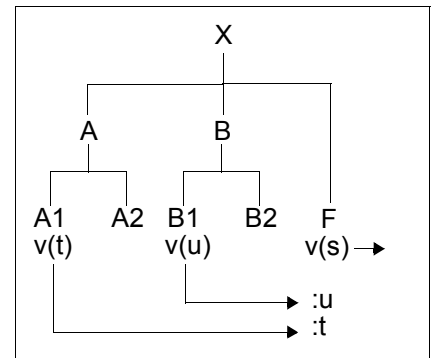
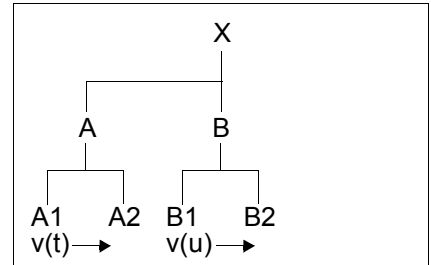
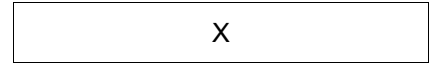
## Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1.
- (3) Program library LIB4 is searched for all unresolved EXTRNs (default value SYMBOL-NAME=\*ALL). The OMs in program library LIB4 are searched in alphabetical order. The OM F resolves the EXTRNs v(t) and v(u). BINDER includes the OM F at the root of the current LLM. The unresolved external reference v(s) in the OM F is entered in the list of unresolved external references. BINDER searches program library LIB4 for a module that resolves the EXTRN v(s), and finds the OM G. The OM G is included at the root of the current LLM.

## Case 6 (rules 4 and 6)

Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B),TYPE=L
- (3) RESOLVE-BY-AUTOLINK  
LIBRARY=LIB4,TYPE=(L,R)  
SCOPE=EXPLICIT(WITHIN-SUB-LLM=B)

Current LLM  
(work area)

## Explanation

- (1) An LLM with the internal name X is created in the work area.
- (2) The LLMs A and B are included in the current LLM from program library LIB1.
- (3) Program library LIB4 is searched for *all* unresolved EXTRNs of the LLM B (default value SYMBOL-NAME=\*ALL). The OMs in program library LIB4 are searched in alphabetical order. The OM F resolves the EXTRN v(u). BINDER includes the OM F at the root of the current LLM. The unresolved external reference v(s) in the OM F is entered in the list of unresolved external references. The EXTRN v(s) remains unresolved because it lies outside the scope. Although the EXTRN v(t) lies outside the scope it is resolved because the module F was included with the ENTRY t.

### 3.5.3 Handling unresolved external references

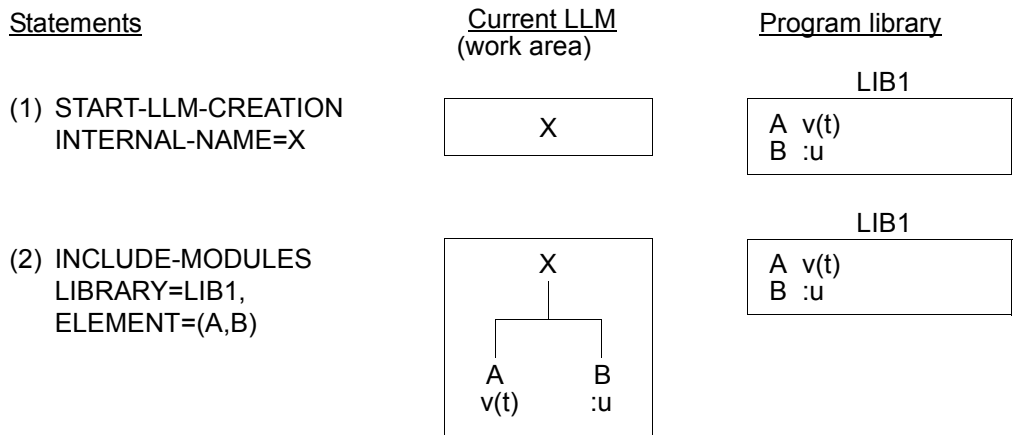
The user uses the SET-EXTERN-RESOLUTION statement to define how BINDER is to handle remaining unresolvable external references for the current LLM. It is possible to define that unresolved external references are valid or invalid. Valid unresolved external references are given the address of a specified symbol.

If unresolved external references are valid, they are taken over when the LLM is saved. If unresolved external references are invalid, the LLM will be rejected when saving is attempted.

The SET-EXTERN-RESOLUTION statement does not take effect until the current LLM is saved with the SAVE-LLM statement. The current LLM in the work area remains unchanged. If modules are included by means of an INCLUDE-MODULES statement between the SET-EXTERN-RESOLUTION statements and the SAVE-LLM statement, and an included module can resolve the unresolved external references, the SET-EXTERN-RESOLUTION statement will be skipped.

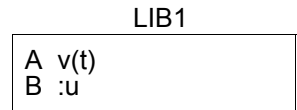
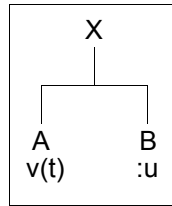
The scope for the handling of unresolved external references can be limited to specific OMs and sub-LLMs in the current LLM (SCOPE operand).

*Example 1*

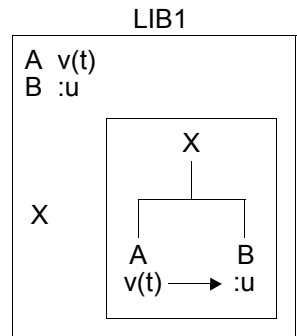
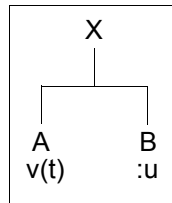




- (3) SET-EXTERN-RESOLUTION  
 SYMBOL-NAME=T,  
 SYMBOL-TYPE=REFERENCES  
 RESOLUTION=BY-SYMBOL  
 (SYMBOL=U)



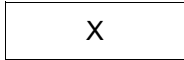
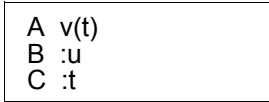
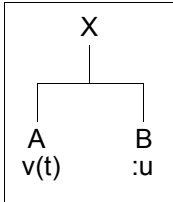
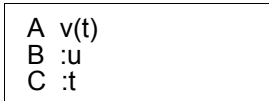
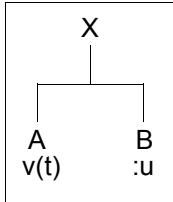
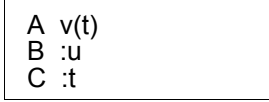
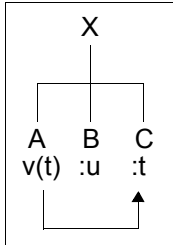
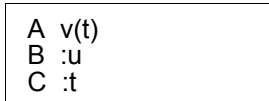
- (4) SAVE-LLM  
 LIBRARY=LIB1



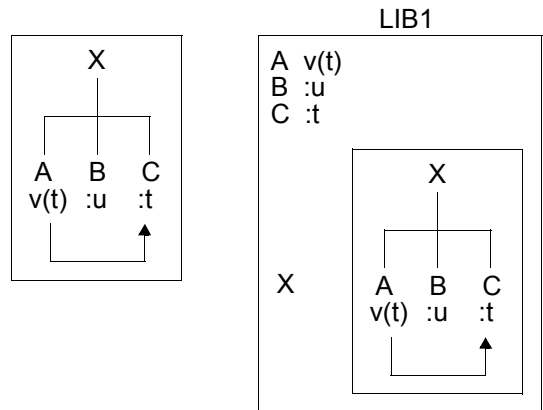
Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The object modules A and B are read from program library LIB1 and included in the current LLM. Object module A contains the unresolved EXTRN v(t), object module B contains the ENTRY u. The EXTRN v(t) remains unresolved because it cannot be resolved by the ENTRY u.
- (3) This defines that the unresolved EXTRN v(t) will be given the address of the ENTRY u when the current LLM is saved. The current LLM in the work area remains unchanged.
- (4) The current LLM is saved as an element with the element name X in program library LIB1. The SET-EXTERN-RESOLUTION statement is executed. The EXTRN v(t) is given the address of the ENTRY u.

Example 2

Statements	Current LLM (work area)	Program library
(1) START-LLM-CREATION INTERNAL-NAME=X		LIB1 
(2) INCLUDE-MODULES LIBRARY=LIB1, ELEMENT=(A,B)		LIB1 
(3) SET-EXTERN-RESOLUTION SYMBOL-NAME=T, SYMBOL-TYPE=REFERENCES RESOLUTION=BY-SYMBOL (SYMBOL=U)		LIB1 
(4) INCLUDE-MODULES LIBRARY=LIB1, ELEMENT=C		LIB1 


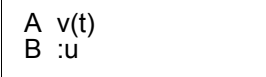
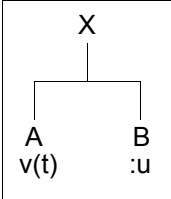
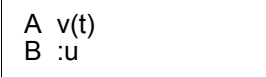
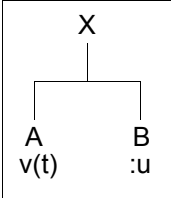

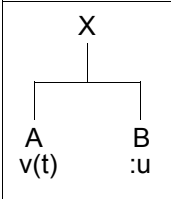
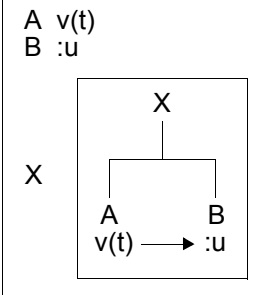
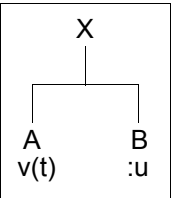
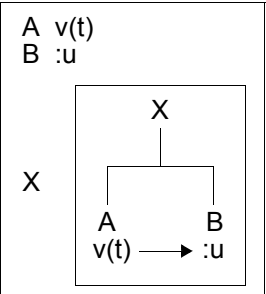
- (5) SAVE-LLM  
LIBRARY=LIB1,  
ELEMENT=X



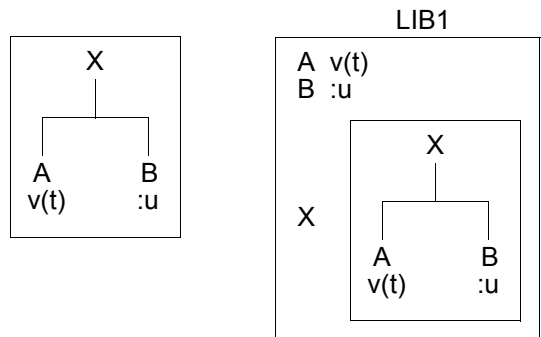
Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The object modules A and B are read from program library LIB1 and included in the current LLM. Object module A contains the unresolved EXTRN v(t), object module B contains the ENTRY u. The EXTRN v(t) remains unresolved because it cannot be resolved by the ENTRY u.
- (3) This defines that the unresolved EXTRN v(t) will be given the address of the ENTRY u when the current LLM is saved. The current LLM in the work area remains unchanged.
- (4) The object module C with the ENTRY t is read from program library LIB1 and included in the current LLM. The ENTRY t can resolve EXTRN v(t).
- (5) The current LLM is saved as an element with the element name X in program library LIB1. Because the EXTRN v(t) was resolved by ENTRY t, the SET-EXTERN-RESOLUTION statement is skipped.

Example 3

Statements	Current LLM (work area)	Program library
(1) START-LLM-CREATION INTERNAL-NAME=X		<p style="text-align: center;">LIB1</p> 
(2) INCLUDE-MODULES LIBRARY=LIB1, ELEMENT=(A,B)		<p style="text-align: center;">LIB1</p> 
(3) SET-EXTERN-RESOLUTION SYMBOL-NAME=T, SYMBOL-TYPE=REFERENCES RESOLUTION=BY-SYMBOL (SYMBOL=U)		<p style="text-align: center;">LIB1</p> 
(4) SAVE-LLM LIBRARY=LIB1		<p style="text-align: center;">LIB1</p> 
(5) START-LLM-UPDATE LIBRARY=LIB1 ELEMENT=X		<p style="text-align: center;">LIB1</p> 

(6) SAVE-LLM  
LIBRARY=LIB1



Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The object modules A and B are read from program library LIB1 and included in the current LLM. Object module A contains the unresolved EXTRN v(t), object module B contains the ENTRY u. The EXTRN v(t) remains unresolved because it cannot be resolved by the ENTRY u.
- (3) This defines that the unresolved EXTRN v(t) will be given the address of the ENTRY u when the current LLM is saved. The current LLM in the work area remains unchanged.
- (4) The current LLM is saved as an element with the element name X in program library LIB1. The SET-EXTERN-RESOLUTION statement is executed. The EXTRN v(t) is given the address of the ENTRY u.
- (5) The same LLM with the element name X in program library LIB1 is updated.
- (6) The current LLM is saved as an element with the element name X in program library LIB1. The specifications under (3) in the SET-EXTERN-RESOLUTION statement are then ignored. The LLM is saved with the EXTRN v(t) unresolved (default value in the SET-EXTERN-RESOLUTION statement).

### 3.6 Handling name conflicts

Name conflicts can occur when multiple entries having the same name in the External Symbols Vector (ESV) of the LLM. Not every instance of identical names is a name conflict, however.

The following table illustrates how BINDER reacts when identical names are encountered.

Entry 2	Entry 1			
	CSECT	ENTRY	COMMON	XDSEC-D
CSECT	(1)	(1)	(2)	--
ENTRY	(1)	(1)	(1)	--
COMMON	(2)	(1)	(3)	--
XDSEC-D	--	--	--	(4)

#### *Explanation*

- (1) A name conflict has been detected. BINDER accepts the name conflict and attempts to resolve it. In doing so it proceeds in accordance with the rules described for the resolution of external references (see [page 80ff](#)).

The user can find information about name conflicts in the lists that are output by the SHOW-MAP statement (see [page 141ff](#)). The user can then take action to resolve the name conflict by renaming symbols or modifying the masking of symbols (see [page 105ff](#)).

- (2) BINDER selects a suitable size for the COMMON that will accommodate the longest CSECT of this name or the longest COMMON of this name.
- (3) BINDER selects a suitable size for the COMMON that will accommodate the longest COMMON.
- (4) A name conflict has been detected. BINDER proceeds as described in point (1). Definitions of XDSECs *cannot* however be renamed or masked by the user.

In the statements

INCLUDE-MODULES,  
MERGE-MODULES,  
MODIFY-MODULE-ATTRIBUTES,  
MODIFY-SYMBOL-VISIBILITY,  
RENAME-SYMBOLS,  
REPLACE-MODULS,  
RESOLVE-BY-AUTOLINK and  
SAVE-LLM

the NAME-COLLISION operand can be used to control the handling of a name conflict caused by the statement in which this operand is specified. Note, however, that any other name conflicts which may occur are not affected by this.

### 3.7 COMMON promotion

COMMONs are sections that at the time of linking do not as yet contain any data or instructions but simply reserve space for that purpose. These areas can be used after loading of the LLM as data communication areas between different modules of the LLM or as reserved space for CSECTs.

BINDER assigns COMMONs of the same name *one* shared storage area. BINDER selects a suitable size for this area that will accommodate the longest COMMON or longest CSECT that promotes a COMMON of this name. After they have been promoted, the COMMONs have the same load address as the CSECT that promoted them.

If the LLM is single slice or sliced by attribute, all the COMMONs with the same name are promoted by the first CSECT with the same name found during the scan of the LLM structure tree from left to right.

If the LLM is sliced by user, all the COMMONs with the same name are sliced by slice. In each slice, these COMMONs are promoted by the first CSECT with the same name included in the slice and found during the scan of the LLM structure tree from left to right.

If the LLM has no External Symbols Vector, a COMMON that has not already been promoted cannot be promoted at load time.

Unnamed COMMONs are handled by BINDER in the same way as named areas except that it does not compare the name field with the names of CSECTs. Text in unnamed CSECTs is not therefore used for promotion of a COMMON.

## 3.8 Handling pseudo-registers

Pseudo-registers are main memory areas that are used for intercommunication between different program sections. The language processors compute the alignment and length of the pseudo-registers and pass this information on to BINDER in the form of ESV information. BINDER combines the pseudo-registers in the modules to form **pseudo-register vectors** and computes the alignment and maximum length of the pseudo-register vectors. The maximum length of a pseudo-register vector may not exceed 4096 bytes.

BINDER does not reserve the main memory area for the pseudo-register vectors. The user must reserve the necessary main memory area.

## 3.9 Address relocation

Each module that is included in the LLM comprises one or more CSECTs with addresses relative to the beginning of the associated module. BINDER assigns relative addresses to the individual CSECTs on linking the LLM. The address of the first CSECT of the LLM is used as the reference address. All other CSECTs in the modules that are included in the LLM contain an address relative to this reference address. In addition, all address references in the CSECTs are adjusted to their relative position. If a CSECT has the relative position 300, for example, relative to the address of the first CSECT in the LLM, all address references in the CSECT are incremented by 300.

If the specified load address is greater than or less than the reference address of the LLM, DBL defines the absolute addresses of the LLM by determining an address constant from the sum of the reference address and the load address and adding this to the relative addresses of the CSECTs.



## 3.10 Handling symbols

**Program definitions** and **references** in an LLM are combined under the generic term **symbols** in the following. Each symbol is identified by means of its name.

**Program definitions** are:

- control sections (CSECTs)
- entry points (ENTRYs)
- COMMONs
- external dummy sections as definitions (XDSEC-Ds)

**References** are:

- external references (EXTRNs)
- V-type constants
- weak external references (WXTRNs)
- external dummy sections as references (XDSEC-Rs)

BINDER handles symbols with the following functions:

- renaming symbols (RENAME-SYMBOLS),
- modifying the attributes of symbols (MODIFY-SYMBOL-ATTRIBUTES),
- modifying the masking of symbols (MODIFY-SYMBOL-VISIBILITY),
- modifying the symbol type (MODIFY-SYMBOL-TYPE).

### 3.10.1 Symbol names

In addition to the usual EN name (external name), BINDER also supports another type of symbol name: the EEN name (extended external name).

- EN names have a maximum length of 32 characters and may contain only the following characters:

A-Z, a-z, 0-9, @, #, \$, \_, &, %, -

The name must not start with a % character.

- EEN names can consist of up to 32723 characters, on which there are no restrictions (nonprintable characters are also permitted). Names of this kind are necessary for compilers of object-oriented programming languages (e.g. C/C++ V3.0) in particular.

Special measures must be taken so that these names can be input and output at user interfaces:

- EEN names cannot be input at the BINDER user interfaces. They can only be directly generated by corresponding compilers and contained in the LLMs generated by these compilers.
- EEN names are output in lists as follows:

BINDER restricts the printable length for the output of EEN names to 32 characters. The compiler which has generated the name provides an algorithm, which BINDER uses to determine a printable string representing the EEN name in its full length. By its nature, this string is not unique and should be used only for information purposes. In particular, a certain EEN name may be presented differently by another application if it determines a different printable length. This kind of printable name should therefore never be used as an input parameter.

#### *Notes*

- The type of a symbol name cannot be changed.
- Both types of symbol name can occur next to one another in the same LLM.
- The type of a symbol name is irrelevant in tests for the same name.
- BLSSERV is required to load LLMs that contain EEN names. Therefore, they cannot be loaded in BS2000 versions earlier than V3.0, and only in BS2000 V3.0, if at least version V2.0 of BLSSERV is used.

### 3.10.2 Renaming symbols

The names of program definitions and references in an LLM can be changed by means of the RENAME-SYMBOLS statement.

The following program definitions can be renamed:

- control sections (CSECTs)
- entry points (ENTRYs)
- COMMONs

The following references can be renamed:

- external references (EXTRNs)
- V-type constants
- weak external references (WXTRNs)

Program definitions and references can be renamed *simultaneously*. The SYMBOL-TYPE operand determines which type is renamed.

*Masked* symbols can also be renamed by means of the RENAME-SYMBOLS statement. External references relating to the masked symbols are deresolved during renaming if requested in the MODIFY-SYMBOL-VISIBILITY statement.

The scope for renaming can be limited to specific OMs and sub-LLMs in the current LLM.

The handling of any name conflicts which may occur can be controlled with the NAME-COLLISION operand.

*Example 1*

This example illustrates the renaming of ENTRYs in an LLM. Here, one object module is excluded.

<u>Statements</u>	<u>Current LLM</u> (work area)
(1) START-LLM-CREATION INTERNAL-NAME=X	X
(2) INCLUDE-MODULES LIBRARY=LIB, ELEMENT=(A,B,C)	<pre> graph TD     X --- A     X --- B     X --- C     X --- D     X --- E     D --- exa1[":exa"]     E --- exa2[":exa"] </pre>
(3) INCLUDE-MODULES LIBRARY=LIB, ELEMENT=(D,E)	<pre> graph TD     X --- A     X --- B     X --- C     X --- D     X --- E     D --- exa1[":exa"]     E --- exa2[":exa"] </pre>
(4) RENAME-SYMBOLS SYMBOL-NAME=EXA, SYMBOL-TYPE=ENTRY, SCOPE=EXPLICIT(EXCEPT- SUB-LLM=X.D), NEW-NAME=ABC	<pre> graph TD     X --- A     X --- B     X --- C     X --- D     X --- E     D --- exa[":exa"]     E --- abc[":abc"] </pre>

## Explanation:

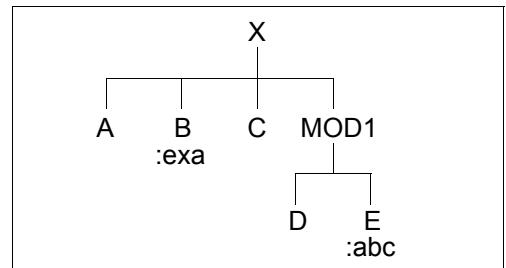
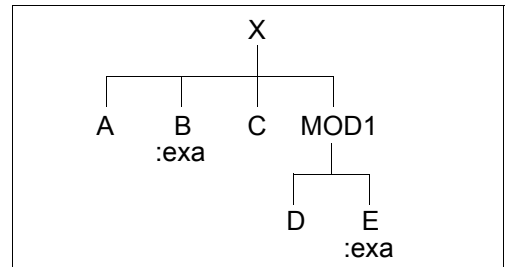
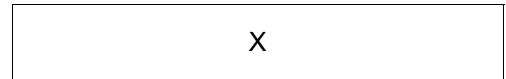
- (1) An LLM with the internal name X is created in the work area.
- (2) The OMs A, B and C are included in the current LLM from program library LIB.
- (3) The OMs D and E are included in the current LLM from program library LIB. D and E have ENTRYs with the name “exa”.
- (4) Only the name “exa” of the ENTRY in module E is changed to “abc”. The name “exa” of the ENTRY in module D remains unchanged because it was excluded by the SCOPE operand.

*Example 2*

This example illustrates the renaming of ENTRYs in an LLM. Here, only one specific sub-LLM is involved.

Statements

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES  
LIBRARY=LIB,  
ELEMENT=(A,B,C)
- (3) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=MOD1
- (4) INCLUDE-MODULES  
LIBRARY=LIB,  
ELEMENT=(D,E)
- (5) RENAME-SYMBOLS  
SYMBOL-NAME=EXA,  
SYMBOL-TYPE=ENTRY,  
SCOPE=EXPLICIT(WITHIN-SUB-  
LLM=X.MOD1),  
NEW-NAME=ABC
- (6) END-SUB-LLM-STATEMENTS

Current LLM  
(work area)

## Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) The OMs A, B and C are included in the current LLM from program library LIB. B has an ENTRY with the name "exa".
- (3) A sub-LLM with the name MOD1 is begun at the root.
- (4) The OMs D and E are included in the current sub-LLM from program library LIB. E also has an ENTRY with the name "exa".
- (5) Only the name "exa" of the ENTRY in module E is changed to "abc" because renaming was restricted to the sub-LLM MOD1 by means of the SCOPE operand. The name "exa" of the ENTRY in module B remains unchanged.
- (6) The current sub-LLM is ended.

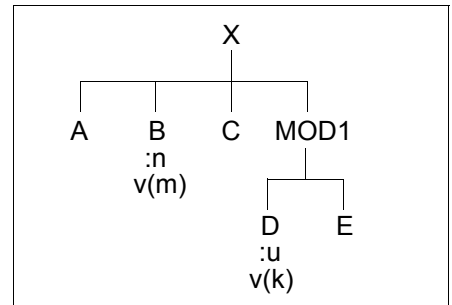
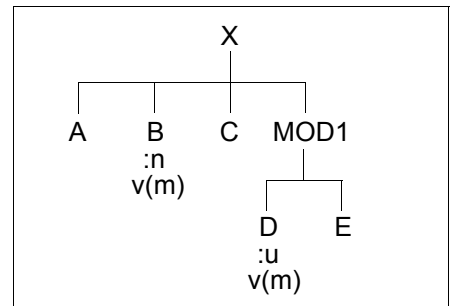
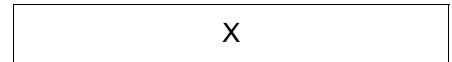
*Example 3*

This example illustrates the renaming of EXTRNs in an LLM. Here, only one specific sub-LLM is involved.

Statements

Current LLM  
(work area)

- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) INCLUDE-MODULES  
LIBRARY=LIB,  
ELEMENT=(A,B,C)
- (3) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=MOD1
- (4) INCLUDE-MODULES  
LIBRARY=LIB,  
ELEMENT=(D,E)
- (5) RENAME-SYMBOLS  
SYMBOL-NAME=M,  
SYMBOL-OCCURENCE=PARAMETERS  
(OCCURENCE-NUMBER=ALL),  
SYMBOL-TYPE=REFERENCES,  
SCOPE=EXPLICIT(WITHIN-SUB-  
LLM=X.MOD1),  
NEW-NAME=K
- (6) END-SUB-LLM-STATEMENTS



Explanation:

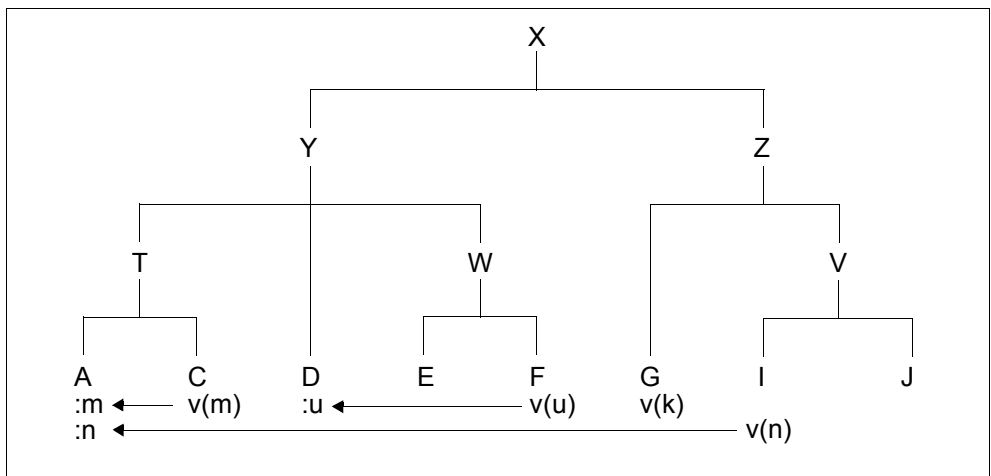
- (1) An LLM with the internal name X is created in the work area.
- (2) The OMs A, B and C are included in the current LLM from program library LIB. B has an ENTRY with the name “n” and an EXTRN with the name “m”.
- (3) A sub-LLM with the name MOD1 is begun at the root.
- (4) The OMs D and E are included in the current sub-LLM from program library LIB. D has an ENTRY with the name “u” and an EXTRN with the name “m”.

- (5) The name “m” of references is to be changed to “k” (SYMBOL-TYPE=REFERENCES operand). Only the name “m” of the EXTRN in module D is changed to “k” because renaming was restricted to the sub-LLM MOD1 by means of the SCOPE operand. Renaming applies to every occurrence of the name (OCCURRENCE-NUMBER=ALL operand).
- (6) The current sub-LLM is ended.

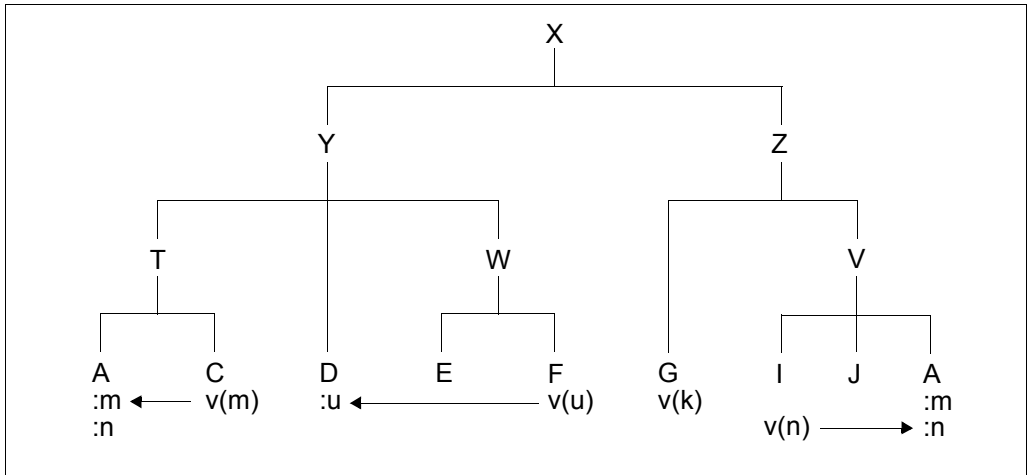
*Example 4*

This example illustrates how the resolution of external references is affected by the renaming of program definitions.

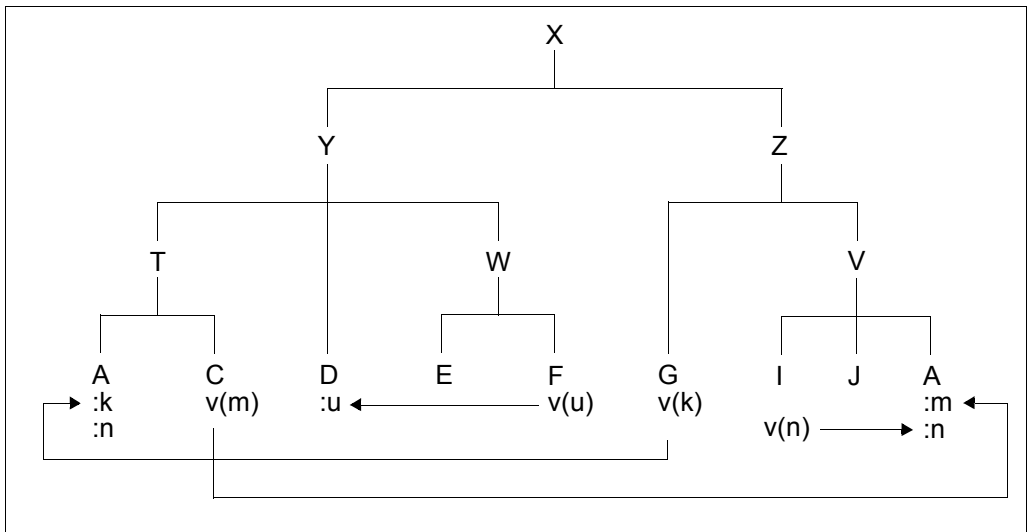
- (1) START-LLM-UPDATE LIBRARY=LIB,ELEMENT=LLM1



(2) INCLUDE-MODULES LIBRARY=LIB2,ELEMENT=A,PATH-NAME=X.Z.V

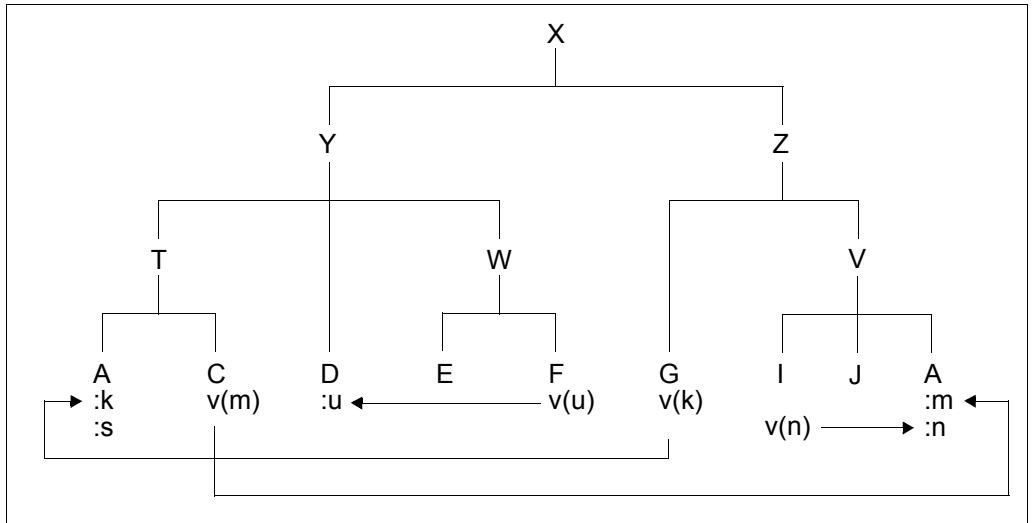


(3) RENAME-SYMBOLS SYMBOL-NAME=M,SYMBOL-TYPE=DEFINITIONS, SCOPE=EXPLICIT(WITHIN-SUB-LLM=X.Y.T.A),NEW-NAME=K





- (4) RENAME-SYMBOLS SYMBOL-NAME=N,SYMBOL-TYPE=DEFINITIONS, SCOPE=EXPLICIT(WITHIN-SUB-LLM=X.Y.T.A),NEW-NAME=S



Explanation:

- (1) An LLM saved as an element with the name LLM1 in program library LIB is to be updated. The LLM contains the following program definitions and references:
  - the EXTRN v(m) in the OM C resolved by ENTRY “m” in the OM A
  - the EXTRN v(u) in the OM F resolved by ENTRY “u” in the OM D
  - the unresolved EXTRN v(k) in the OM G
  - the EXTRN v(n) in the OM I resolved by ENTRY “n” in the OM A.
- (2) The same object module A that is already included in the sub-LLM T is read from program library LIB2 and included in the sub-LLM V. The ENTRY “n” in the most recently included OM A now resolves the EXTRN v(n) in the OM I because OM A lies in the same sub-LLM as OM I.
- (3) The ENTRY “m” in the OM A of the sub-LLM T is renamed “k”. The external references are thus resolved as follows:
  - EXTRN v(k) in the OM G is resolved by ENTRY “k” in the OM A of the sub-LLM T
  - ENTRY v(m) in the OM C is resolved by ENTRY “m” in the OM A of the sub-LLM V.
- (4) The ENTRY “n” in the OM A of the sub-LLM T is renamed “s”. This has no influence on the resolution of the external references.

### 3.10.3 Modifying the attributes of symbols

The attributes of control sections (CSECTs) and COMMONs in the current LLM can be changed with the MODIFY-SYMBOL-ATTRIBUTES statement.

The following attributes can be modified (see [page 18ff](#)):

- main memory resident (RESIDENT)
- shareable (PUBLIC)
- read access (READ-ONLY)
- alignment (ALIGNMENT)
- addressing mode (AMODE)
- residence mode (RMODE)

When modifying the attributes it should be noted that all COMMONs must have the same name and all CSECTs of the same name that initialize these COMMONs with data must have the same value for the READ-ONLY attribute.

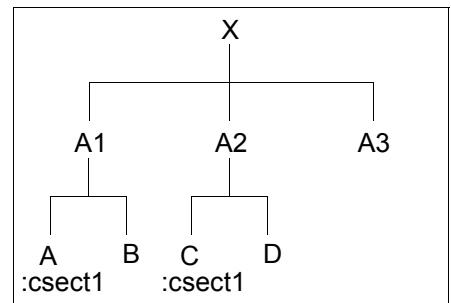
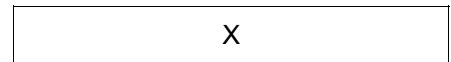
The scope for the modification of attributes can be limited to specific OMs and sub-LLMs in the current LLM (SCOPE operand).

*Example*

#### Statements

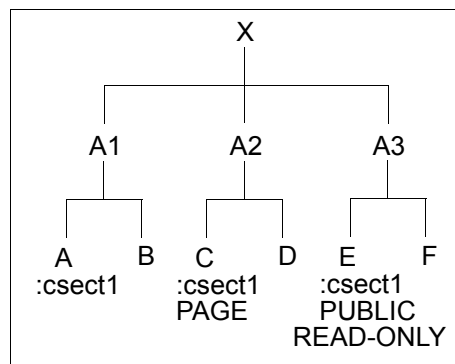
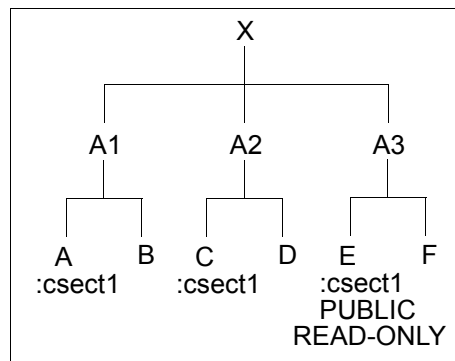
- (1) START-LLM-CREATION  
INTERNAL-NAME=X
- (2) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=A1
- (3) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(A,B)
- (4) END-SUB-LLM-STATEMENTS
- (5) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=A2
- (6) INCLUDE-MODULES LIBRARY=LIB1,  
ELEMENT=(C,D)
- (7) END-SUB-LLM-STATEMENTS
- (8) BEGIN-SUB-LLM-STATEMENTS  
SUB-LLM-NAME=A3

Current LLM  
(work area)



Statements

- (9) INCLUDE-MODULES  
LIBRARY=LIB1,  
ELEMENT=(E,F)
- (10)MODIFY-SYMBOL-ATTRIBUTES  
SYMBOL-NAME=CSECT1,  
READ-ONLY=YES,  
PUBLIC=YES
- (11)END-SUB-LLM-STATEMENTS
- (12)MODIFY-SYMBOL-ATTRIBUTES  
SYMBOL-NAME=CSECT1,  
SCOPE=EXPLICIT(WITHIN-SUB-  
LLM=X.A2),  
ALIGNMENT=PAGE

Current LLM  
(work area)

## Explanation:

- (1) An LLM with the internal name X is created in the work area.
- (2) A sub-LLM with the name A1 is begun.
- (3) The object modules A and B are included in the current sub-LLM A1 from program library LIB1. Object module A contains a CSECT with the name CSECT1.
- (4) A1 is ended.
- (5) A sub-LLM with the name A2 is begun.
- (6) The object modules C and D are included in A2 from program library LIB1. Object module C contains a CSECT with the name CSECT1.
- (7) A2 is ended.
- (8) A sub-LLM with the name A3 is begun.

- (9) The object modules E and F are included in A3 from program library LIB1. Object module E contains a CSECT with the name CSECT1.
- (10) The PUBLIC and READ-ONLY attributes of CSECT1 are modified. The *current* sub-LLM is addressed. This is the sub-LLM A3. The attributes of the CSECTs "CSECT1" in object modules A and C remain unchanged.
- (11) A3 is ended.
- (12) The ALIGNMENT attribute of the CSECT1 is modified. Object module C in the sub-LLM A2 is addressed with the path name X.A2. The attributes of the CSECTs "CSECT1" in object modules A and E remain unchanged.

### 3.10.4 Modifying the masking of symbols

The user has the facility to mask control sections (CSECTs) and entry points (ENTRYs) in the current LLM. A masked symbol is saved in the External Symbols Vector (ESV) of the LLM, but it is provided with a flag (mask) which makes it "invisible". Such a symbol is thus "invisible" for the autolink function and is not used for the resolution of external references.

The extent to which the symbols remain visible or are masked is determined by the user with the MODIFY-SYMBOL-VISIBILITY statement. With masked symbols it is possible to choose whether resolved external references relating to the specified symbols remain resolved or are to be deresolved (KEEP-RESOLUTION operand).

The scope for modifying the masking of symbols can be limited to specific OMs and sub-LLMs in the current LLM (SCOPE operand).

The handling of any name conflicts which may occur can be controlled with the NAME-COLLISION operand.

*Example*

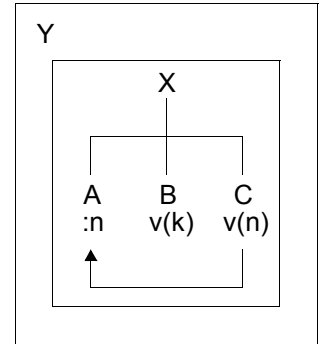
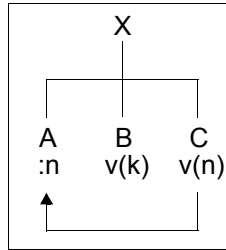
This example illustrates how the resolution of external references is affected by the masking and renaming of symbols.

Statements

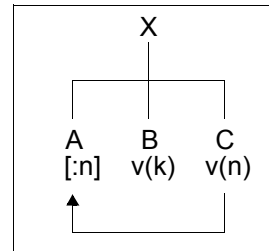
Current LLM  
(work area)

Program library  
(LIB1)

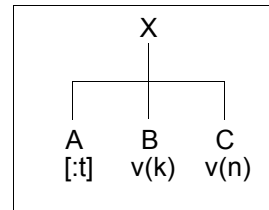
- (1) START-LLM-UPDATE  
LIBRARY=LIB1,  
ELEMENT=Y



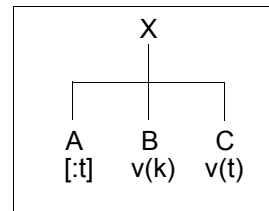
- (2) MODIFY-SYMBOL-VISIBILITY  
SYMBOL-NAME=N,  
VISIBLE=NO (KEEP-  
RESOLUTION=YES)



- (3) RENAME-SYMBOLS  
SYMBOL-NAME=N,  
SYMBOL-TYPE=DEFINITIONS,  
NEW-NAME=T



- (4) RENAME-SYMBOLS  
SYMBOL-NAME=N,  
SYMBOL-TYPE=REFERENCES,  
NEW-NAME=T



Explanation:

- (1) An LLM X saved as an element with element name Y in program library LIB1 is to be updated. The LLM X contains the EXTRN v(n) that is resolved by the ENTRY n, and the unresolved EXTRN v(k).
- (2) The ENTRY n in the LLM X is masked (masking is represented by square brackets). The resolved EXTRN v(n) remains resolved (KEEP-RESOLUTION=YES operand).
- (3) The name “n” of the masked ENTRY is changed to “t”. The resolved EXTRN (n) is thus deresolved.
- (4) The names “n” of the EXTRNs are changed to “t”. The EXTRN v(t) cannot be resolved by the ENTRY t because the ENTRY t is masked.

### 3.10.5 Modifying symbol types

With the MODIFY-SYMBOL-TYPE statement, the user can modify the types of symbols in the current LLM. Note, however, that “symbols” in this context can only be references. The user cannot modify the types of program definitions (CSECTs, ENTRYs).

External references (EXTRNs), V constants (VCONs) and weak external references (WXTRNs) can be converted into each other almost without restrictions. The user can, for example, convert EXTRNs or VCONs into WXTRNs in order to prevent these external references from being resolved by the autolink function. However, it is not possible to convert unreferenced EXTRNs and WXTRNs into VCONs.

A further typical application for the MODIFY-SYMBOL-TYPE statement results from the use of OCM (Overlay Control Module) with user-defined slices (see the START-LLM-CREATION statement). Since only VCONs may be used as references in OCMs, all references must have the symbol type VCON.

The scope for modification of symbol types can be restricted to certain object modules and sub-LLMs in the current LLM (operand SCOPE).

The MODIFY-SYMBOL-TYPE statement has no effect on external references which have already been resolved.

### 3.11 Merging modules

A sub-LLM or an entire LLM can be “merged”, i.e. all CSECTs of the sub-LLM combined to form a single CSECT. External references between the merged modules remain resolved and are deleted from the External Symbols Vector. With the operand ENTRY-LIST, the user can specify which CSECTs and ENTRYs are to remain in the External Symbols Vector, and can thus still be used for resolving external references. Each CSECT which is merged and remains in the External Symbols Vector is converted into an ENTRY.

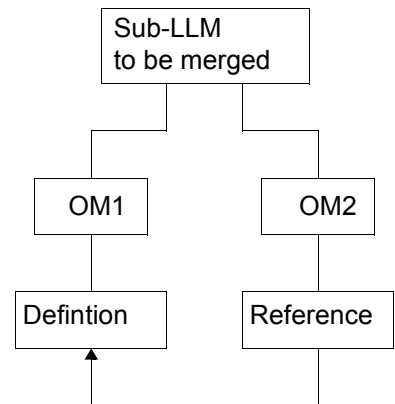
The following examples show three possible situations which can occur when a sub-LLM is merged.

#### *Case 1*

The sub-LLM to be merged contains only references to symbols within the same sub-LLM.

Consequence:

The external reference is finally resolved and is deleted from the External Symbols Vector. If the user specifies the definition in operand ENTRY-LIST, it remains in the ESV and is converted into an ENTRY if it was a CSECT.



OM: object module

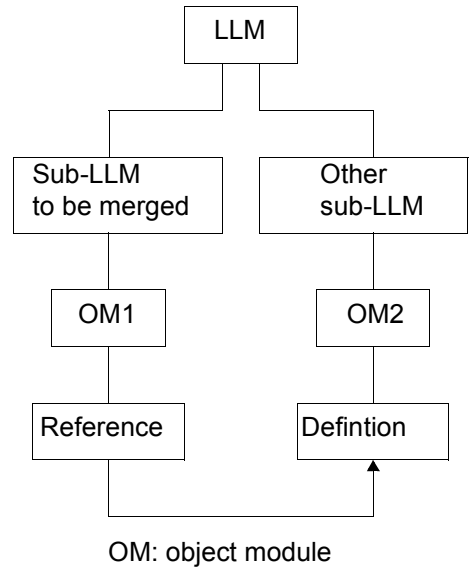
*Case 2*

The sub-LLM to be merged contains references to symbols in one or more other sub-LLMs.

Consequence:

The resolution of external references remains unchanged.

The External Symbols Vector is also not affected.



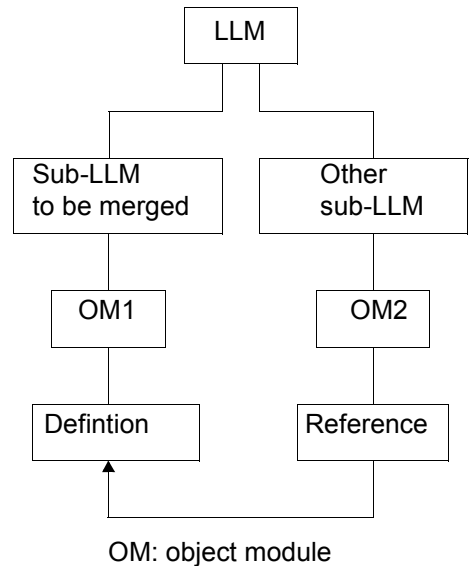
*Case 3*

The sub-LLM to be merged contains definitions to which other sub-LLMs refer.

Consequence:

If the user specifies these definitions in the ENTRY-LIST operand, they remain in the External Symbols Vector and are changed to ENTRYs if they were CSECTs. No changes are made to the resolution of the external reference in OM2.

If, however, the user does not specify the ENTRY-LIST operand, the definition is deleted from the External Symbols Vector. The external reference in OM2 is now unresolved and BINDER attempts to resolve this external reference with the aid of the definitions which exist in the External Symbols





After the merge operation, the following logical structure exists for the sub-LLM:

- one sub-LLM node with the name of the merged sub-LLM,
- one prelinked module (as a “leaf” in the logical structure, see [page 16ff](#)) with the same name as the merged sub-LLM.

All nodes removed by the merge procedure are still visible in the BINDER list during the BINDER run. However, they are not stored with SAVE-LLM.

If the CSECTs to be merged are contained in different slices, the following should be noted:

- LLMs with user-defined slices may not be merged.
- In the case of slices by attributes, the new CSECT is inserted in the slice whose attributes match those specified with the NEW-CSECT-ATTRIBUTES operand.

If a definition specified in ENTRY-LIST exists more than once in the sub-LLM to be merged, only one definition with this name will remain in the External Symbols Vector. This is determined as follows:

- CSECTs have priority over ENTRYs,
- the first definition with this name which occurs in the External Symbols Vector remains there.

No list for symbolic debugging (LSD) is generated for the new CSECT. The merge module is not regarded as a runtime module, which means that the visibility of the symbols also remains unchanged. COMMONs, external dummy sections (XDSEC-D) and pseudo-registers in the merged modules also remain unchanged. The initialization of the COMMONs is also updated, as is the External Symbols Vector.

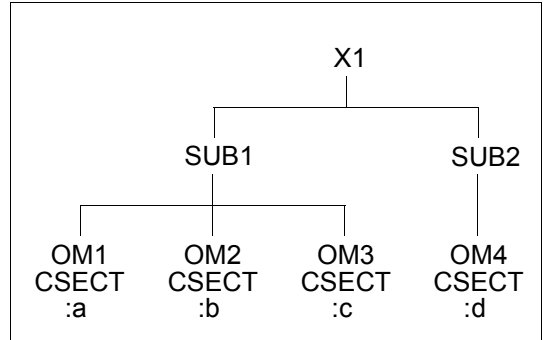
*Example 1*

Merging an entire LLM. The result is an LLM which contains only a single module (prelinked module).

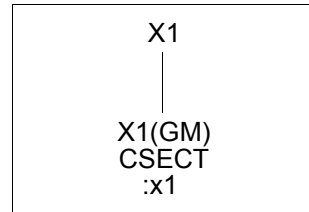
Statements

- (1) START-LLM-UPDATE  
LIBRARY=LIB1,  
ELEMENT=M

Current LLM  
(work area)



- (2) MERGE-MODULES  
NAME=X1,  
PATH-NAME=\*NONE



GM: prelinked module

Explanation:

- (1) An LLM stored as an element with the name M in program library LIB1 is to be merged. For this, it is read into the work area. The LLM has the internal name X1.
- (2) The entire LLM with the internal name X1 is merged, since PATH-NAME=\*NONE was specified. The new CSECT receives the same name as the LLM because the default value \*NAME is used for NEW-CSECT-NAME.

*Example 2*

Merging a sub-LLM with changes to the resolution of the external references.

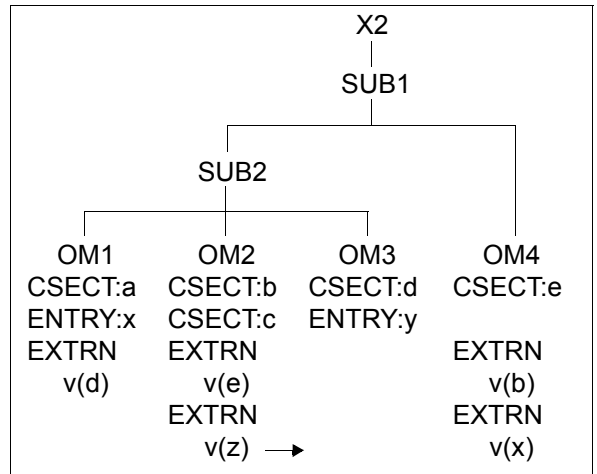
Initial situation:

In LLM X2, all external references except EXTRN v(z) are resolved by CSECTs or ENTRYs in LLM X2.

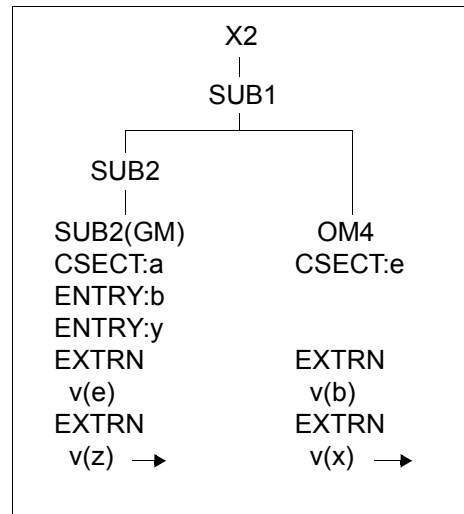
Statements

(1) START-LLM-UPDATE  
LIBRARY=LIB1,  
ELEMENT=N

Current LLM  
(work area)



(2) MERGE-MODULES  
NAME=SUB2,  
PATH-NAME=X2.SUB1,  
NEW-CSECT-NAME=\*STD,  
ENTRY-LIST=(b,y)



GM: prelinked module

Explanation:

- (1) A sub-LLM of the LLM with the internal name X2 is to be merged. For this, the LLM X2, which is stored as library element N in program library LIB1, is read into the work area.
- (2) Sub-LLM SUB2, which can be accessed via path X2.SUB1, is merged. The new CSECT receives the name of the first CSECT in sub-LLM SUB2. The program definitions b and y from sub-LLM SUB2 remain in the External Symbols Vector, whereby b is converted into an ENTRY. All other program definitions from SUB2 (c,d,x) are deleted from the External Symbols Vector (ESV).

After SUB2 has been merged:

- EXTRN v(d) is finally resolved and no longer appears in the ESV
- EXTRN v(z) is still unresolved
- EXTRN v(x) is now also unresolved because it lies outside sub-LLM SUB2 and the ENTRY x in SUB2 has been deleted from the ESV.

## 3.12 Defining default values

The user can define his/her own default values for some of the operands for a BINDER run or an edit run (see [page 138](#)).

The following default values can be defined:

- **MAP defaults:**  
These are defined with the MODIFY-MAP-DEFAULTS statement and are valid for the duration of a BINDER run.
- **CURRENT defaults:**
  - The current input library (CURRENT-INPUT-LIB) for inclusion and replacement of modules is defined with the INCLUDE-MODULES and REPLACE-MODULES statements.
  - The current input/output library for LLMs, the current element name and the current element version are defined with the START-LLM-UPDATE and SAVE-LLM statements.
  - The current sub-LLM is set with the BEGIN-SUB-LLM-STATEMENTS statement. The END-SUB-LLM-STATEMENTS statement makes the next higher LLM the current sub-LLM.
- **INCLUSION-DEFAULTS:**  
These are set in the START-LLM-CREATION, START-LLM-UPDATE and MODIFY-LLM-ATTRIBUTES statements for the duration of an edit run and can be changed temporarily for the duration of an INCLUDE-MODULES or REPLACE-MODULES statement.
- **SAVE defaults:**  
These values are defined in the SAVE-LLM ...=LAST-SAVE... statement and are available for the duration of an edit run.
- **Global defaults (STD-DEFAULTS):**  
Global defaults which define the format of the LLM and the handling of name conflicts can be set with the MODIFY-STD-DEFAULTS statement. They are valid for the duration of a BINDER run.

## 3.13 Display functions

### 3.13.1 Displaying the default values

The SHOW-DEFAULTS statement permits the user to display the current default values (see [page 125](#)). The following defaults can be selected for display:

- the global defaults (STD-DEFAULTS)
- the CURRENT defaults
- the INCLUSION-DEFAULTS
- the values for the last saving of LLMs (LAST-SAVE) or
- the defaults for the BINDER lists (MAP-DEFAULTS).

The CURRENT-DEFAULTS, INCLUSION-DEFAULTS and LAST-SAVE operands are meaningful only after the START-LLM-CREATION or START-LLM-UPDATE statement has been executed. The values are output on SYSOUT.

#### *Example*

The example below illustrates output of the SHOW-DEFAULTS statement.

```

/start-binder
% BND0500 BINDER VERSION 'V02.7A00' STARTED
//start-llm-creation internal-name=complex1
//show-defaults
STD-DEFAULTS:
  OVERWRITE                =YES
  FOR-BS2000-VERSIONS     =FROM-CURRENT
  CONNECTION-MODE         =OSD-DEFAULT
  REQUIRED-COMPRESSSION=NO
  NAME-COLLISION:
    INCLUSION              =IGNORED
    SAVE                   =IGNORED
    SYMBOL-PROCESSING=IGNORED
CURRENT-DEFAULTS:
  CURRENT-SUB-LLM         =COMPLEX1
LIBRARY:
  CURRENT                 =
  CURRENT-INPUT-LIB=
ELEMENT:
  CURRENT                 =

```

```

VERSION:
  CURRENT-VERSION =
INCLUSION-DEFAULTS:
  LOGICAL-STRUCTURE=WHOLE-LLM
  TEST-SUPPORT    =NO
LAST-SAVE:
  OVERWRITE          =STD
  FOR-BS2000-VERSIONS =STD
  REQUIRED-COMPRESSON=STD
  NAME-COLLISION     =STD
  SYMBOL-DICTIONARY  =YES
  RELOCATION-DATA     =YES
  LOGICAL-STRUCTURE  =WHOLE-LLM
  TEST-SUPPORT       =YES
  LOAD-ADDRESS       =UNDEFINED
  ENTRY-POINT        =*STD
  MAP                 =YES
MAP-DEFAULTS:
  MAP-NAME           =*STD
  COMMENT            =NONE
  HELP-INFORMATION   =YES
  GLOBAL-INFORMATION =YES
  LOGICAL-STRUCTURE  =YES
  RESOLUTION-SCOPE   =YES
  PHYSICAL-STRUCTURE =YES
  PROGRAM-MAP        =PARAMETERS
  DEFINITIONS        =ALL
  INVERTED-XREF-LIST=NONE
  REFERENCES         =ALL
  UNRESOLVED-LIST    =SORTED (WXTRN=YES,NOREF=NO)
  SORTED-PROGRAM-MAP =NO
  PSEUDO-REGISTER    =NO
  UNUSED-MODULE-LIST =NO
  DUPLICATED-LIST    =NO
  MERGED-MODULES     =YES
  INPUT-INFORMATION  =YES
  STATEMENT-LIST     =NO
  OUTPUT             =*SYSLST
  SYSLST-NUMBER      =STD
  LINES-PER-PAGE     =64
  LINE-SIZE          =136
//end
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'OK'

```

### 3.13.2 Displaying symbol information

With the SHOW-SYMBOL-INFORMATION statement, the user can display information about symbols. The amount of information to be displayed can be restricted with the INFORMATION operand, depending on what the user wishes to see at the moment. The user can, for example, display the positions of symbols in the logical structure of the LLM, their attributes or their relative addresses (name of the slice and relative address within this slice) either separately or simultaneously. The user can also select the following for display:

- all visible definitions
- the COMMONs and their initialization
- the resolved external references
- a list of symbol names used more than once
- a list of the unresolved external references.

The type of display generally corresponds to the type used for the BINDER lists (see [page 141ff](#)). The information is output on SYSOUT.



*Example*

```

/start-binder
% BND0500 BINDER VERSION 'V02.7A00' STARTED
//start-llm-update library=bnd.llmlib,element=complex1 _____ (1)
//show-symbol-information information=all _____ (2)
AUTOA SD _____ (3)
  @=000000A8 L=00000040 (AA.....)
  IN MODULE : AUTOA
  IN SLICE:PUU-ROU-RTU-RMU + 000000A8
AUTOA SD
  @=00000068 L=00000040 (AA.....)
  IN MODULE : AUTOA
  IN SLICE:PUU-ROU-RTU-RMU + 00000068
AUTOA SD
  @=00000000 L=00000040 (AA.....)
  IN MODULE : AUTOA
  IN SLICE:PUU-ROU-RTU-RMU + 00000000
AUTO2 CM
  NOT PROMOTED L=00000050 (AA.....)
  IN MODULE : AUTO2
AUTO2 CM
  NOT PROMOTED L=00000050 (AA.....)
  IN MODULE : AUTO2
AUTO2 SD
  @=000000E8 L=00000018 (AA.....)
  IN MODULE : AUTO2
  IN SLICE:PUU-ROU-RTU-RMU + 000000E8
AUTO2 SD
  @=00000040 L=00000018 (AA.....)
  IN MODULE : AUTO2
  IN SLICE:PUU-ROU-RTU-RMU + 00000040
AUTO22 SD
  @=00000118 L=00000006 (AA.....)
  IN MODULE : AUTO22
  IN SLICE:PUU-ROU-RTU-RMU + 00000118
AUTO22 CM
  @=00000118 L=00000050 (AA.....)
  IN MODULE : AUTO23
AUTO2 SD
  @=00000100 L=0000000C (AA.....)
  IN MODULE : AUTO2
  IN SLICE:PUU-ROU-RTU-RMU + 00000100
AUTO2 SD
  @=00000058 L=0000000C (AA.....)
  IN MODULE : AUTO2
  IN SLICE:PUU-ROU-RTU-RMU + 00000058
AUTO21 SD

```

```
@=00000110      L=00000006      (AA.....)
IN MODULE : AUTO21
IN SLICE:PUU-ROU-RTU-RMU + 00000110
AUTO23          SD
@=00000168      L=00000006      (AA.....)
IN MODULE : AUTO23
IN SLICE:PUU-ROU-RTU-RMU + 00000168
//end
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'OK'
```

- (1) LLM COMPLEX1 is read into the BINDER work area.
- (2) The following information is output for all symbols in the LLM:
  - Symbol name
  - Type of symbol (CSECT, COMMON,...)
  - Load address for CSECTs and ENTRIES; for COMMONs, the load address of the CSECT that promoted the COMMON
  - Length of the text information in the case of CSECTs
  - Attributes
  - Name of the module containing the symbol
  - Name of the slice containing the symbol and relative address of the symbol in the slice
- (3) The first symbol with the name AUTOA is a CSECT and has the load address 000000A8 (length of the text information of the CSECT: 00000040). The CSECT has the following attributes: AMODE=ANY, RMODE=ANY. No other attributes are specified. The symbol is located in the module AUTOA and in the slice with the name PUU-ROU-RTU-RMU. There it has the relative address 000000A8.

### 3.13.3 Displaying and checking library elements

LLMs or object modules in specific libraries can be displayed and checked at regular intervals with the SHOW-LIBRARY-ELEMENTS statement. The user can, just as in the list output function SHOW-MAP, select the output device for the information. In order to avoid name conflicts, the user can have a list of endangered symbols (DUPLICATE SYMBOLS) generated during a BINDER run. The generated lists (except for those output on SYSLST) are by default ISAM files with ISAM keys with a length of 8.

The ISAM key can be used for evaluation of the lists. The ISAM key is described in the appendix ([page 369f](#)).

#### Example

In the following example the user first has information about the LLMs AUTOLINKL and AUTOLINKR output from the BND.LLMLIB program library to SYSLST. After this the user requests a list of the symbols with the same names for these two LLMs (DUPLICATE SYMBOLS) and also has this list output to SYSLST.

```

/start-binder
% BND0500 BINDER VERSION 'V02.7A00' STARTED
//show-library-elements library=bnd.llmlib, -
//                      element=(autolinkl,autolinkr) _____ (1)
//show-library-elements library=bnd.llmlib,-
//                      element=(autolinkl,autolinkr),-
//                      select=name-collision _____ (2)
%//end
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'OK'

```

(1) Information about the library elements AUTOLINKL and AUTOLINKR is displayed.

```

BINDER V02.7A *LIBRARY CONTENT* DATE=2016-02-03 10:43:51 PAGE 1
LIBRARY LINKNAME TYPE
TYP ELEMENT VERSION
TY SYMBOL TY SYMBOL TY SYMBOL TY SYMBOL TY SYMBOL
-----
:CTID:$USERID.BND.LLMLIB PLAM
(L) AUTOLINKL @
SD AUTOA SD AUTOC SD AUTO2 SD AUTOA SD AUTOA
SD AUTOC SD AUTO2
(L) AUTOLINKR @
SD AUTO21 SD AUTO22 SD AUTO23

```

--- END OF SECTION ---

- (2) The list of symbols with the same name is output for library elements AUTOLINKL and AUTOLINKR:

```

BINDER V02.7A   *DUPLICATE SYMBOLS*   DATE=2016-02-03 10:43:51 PAGE    1
SYMBOL
  LIBRARY                LINKNAME TYPE
  TYP ELEMENT  VERSION
  SYMBOL TYPE
-----
AUTOA
  :CTID:$USERID.BND.LLMLIB          PLAM
  (L) AUTOLINKL @
  CSECT
  CSECT
  CSECT
AUTOB
  :CTID:$USERID.BND.LLMLIB          PLAM
  (L) AUTOLINKL @
  CSECT
  CSECT
AUTO2
  :CTID:$USERID.BND.LLMLIB          PLAM
  (L) AUTOLINKL @
  CSECT
  CSECT

```

--- END OF SECTION ---

## 3.14 Controlling logging

Lists containing information about the current LLM are output:

- by means of the SHOW-MAP statement
- by means of the SAVE-LLM statement (MAP operand) on saving an LLM.

The following may be selected as the output destination:

- the system file SYSLST.
- a file that is output to SYSOUT automatically by means of an implicit SHOW-FILE command (see the “Commands” manual [5]). This file is created under the default file name and deleted immediately after it has been output unless the user prevents this.
- a file whose name is specified by the user.
- a file defined by the default file link name or a specified file link name.
- a user-written subroutine which evaluates the supplied information. This information is output to an ISAM file which the subroutine must access.

These output destinations are also available in the SHOW-LIBRARY-ELEMENTS statement.

Operands in the SHOW-MAP statement determine which information is output. Default settings can be modified with the MODIFY-MAP-DEFAULTS statement. The various lists are described on [page 141f](#).

The START-STATEMENT-RECORDING and STOP-STATEMENT-RECORDING statements are used to activate and deactivate recording of BINDER statements. The statements recorded in this way are output in a list using the SHOW-MAP statement if STATEMENT-LIST operand is set to YES.

## 3.15 Controlling error processing

### 3.15.1 Severity classes

BINDER uses a table of **severity classes** for error processing purposes. These classes describe all the possible actions to be taken by BINDER's error processing routines if errors occur in the BINDER run. The following severity classes are possible:

Severity class	Meaning
INFORMATION	No error detected. An information messages output.
WARNING	It is possible that the LLM cannot be loaded. A warning message is output..
UNRESOLVED EXTERNS	The LLM contains unresolved external references. Loading is possible.
SYNTAX ERROR	Error detected during syntax checking of a statement.
RECOVERABLE ERROR	Error detected. With some types of error, processing is continued automatically or upon input of an acknowledgment.
FATAL ERROR	Error detected. Processing of the LLM is aborted without output of a dump.
INTERNAL ERROR	Internal error. Processing of the LLM is aborted. The user can request that a dump be output.

Each severity class has a certain weight. The INFORMATION class has the lowest severity level, the INTERNAL ERROR class the highest severity level. The severity classes to be handled by BINDER are defined by the user with the MODIFY-ERROR-PROCESSING statement.

The user can define for the severity classes whether, upon the occurrence of errors of certain severity classes,

- the BINDER run is to be terminated
- error messages are to be output
- user and/or task switches are to be set.

### 3.15.2 Message handling

Using the MESSAGE-CONTROL operand in the MODIFY-ERROR-PROCESSING statement it is possible to define for which *severity classes* messages are to be output. This operand determines the *lowest* severity class as of which messages are output. The following table illustrates how the MESSAGE-CONTROL operand controls message output.

Messages of severity class	MESSAGECONTROL		
	INFORMATION	WARNING	ERROR
INFORMATION	yes	no	no
WARNING	yes	yes	no
UNRESOLVED EXTERNS	yes	yes	no
SYNTAX ERROR	yes	yes	yes
RECOVERABLE ERROR	yes	yes	yes
FATAL ERROR	yes	yes	yes
INTERNAL ERROR	yes	yes	yes





## 4 BINDER input/output

Figure 7 provides an overview of the possible BINDER input and output files.

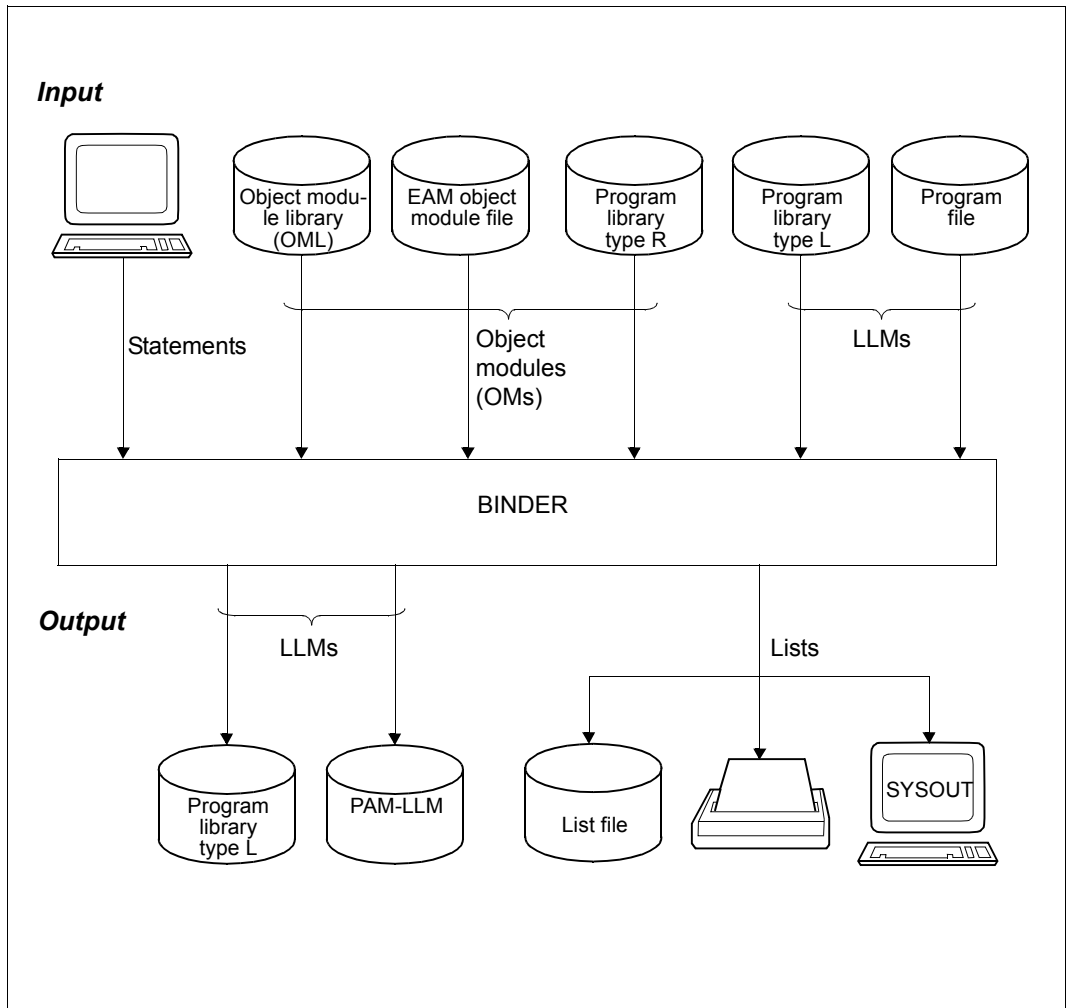


Figure 7: BINDER input and output files

## 4.1 Inputs for BINDER

The inputs for BINDER are statements and modules.

### Statements

BINDER expects statements from the system file SYSDDTA. The statements can be entered interactively or from procedure files. The **dialog interface SDF** (System Dialog Facility) is available to the user for interactive input (see [page 187ff](#)). Each statement is processed *immediately* after it is entered.

Operand values can be specified as global values in the BINDER statements. These values can then be used as default values within a defined scope in the subsequent statements. The following scopes are defined:

- Scope for *one* statement  
This scope is local. An operand value applies only to the associated statement.
- Scope for an edit run  
An **edit run** comprises a sequence of statements that begins with the START-LLM-CREATION or START-LLM-UPDATE statement and ends with the next START-LLM-CREATION or START-LLM-UPDATE statement or with the END statement.  
An operand value applies to an edit run if it can be used as a default value in other statements of the same edit run.
- Scope for a BINDER run  
A **BINDER run** is a sequence of statements that begins after the BINDER load call and ends with the END statement.  
An operand value applies to a BINDER run if it can be used as a default value in other statements of the same BINDER run.

*Example*

*BINDER-run*

```

      START-BINDER _____ (1)
      START-LLM-CREATION INTERNAL-NAME=X
      INCLUDE-MODULES ...
      .
      .
      SAVE-LLM LIBRARY=LIB1,ELEMENT=*CURRENT-NAME_____ (2)
      INCLUDE-MODULES ...
      .
      .
      SAVE-LLM LIBRARY=*CURRENT,ELEMENT=A1_____ (3)
      INCLUDE-MODULES ...
      .
      .
      SAVE-LLM LIBRARY=*CURRENT,ELEMENT=*CURRENT-NAME_____ (4)

      START-LLM-UPDATE LIBRARY=LIB2,ELEMENT=B
      REPLACE-MODULES ...
      .
      .
      SAVE-LLM LIBRARY=*CURRENT,ELEMENT=B_____ (5)
      END
  
```

First edit run

Second edit run

## Explanation:

- (1) BINDER load call
- (2) The internal name X is taken over as the element name from the START-LLM-CREATION statement of the same edit run.
- (3) The program library LIB1 is taken over from the last SAVE-LLM statement of the same edit run.
- (4) The program library LIB1 and the element name A1 are taken over from the last SAVE-LLM statement of the same edit run.
- (5) The program library LIB2 is taken over from the START-LLM-UPDATE statement of the same edit run.

## Modules

Modules can be object modules (OMs), link and load modules (LLMs) or both. The following are input sources for modules:

- for OMs: a program library (element type R), an object module library (OML) or the EAM object module file,
- for LLMs: a program library (element type L).

## Opening the input sources

Program libraries, module libraries and the EAM object module file are opened for input in the order in which they are specified in the statement sequence. They remain open during the entire BINDER run and are closed only by the END statement. An input library cannot therefore be used simultaneously as an output library. For all files that are read in input mode BINDER uses the DMS “secondary read” function (see the “Introductory Guide to DMS” [15]).

## Simultaneous BINDER runs

All input sources accessed in BINDER statements are opened in read-only mode, which means that write accesses (as in SAVE-LLM) are forbidden. This means that several tasks may access one and the same LLM. Several tasks can, for example, transfer the same LLM to the BINDER work area with START-LLM-UPDATE and modify it there. However, a task may store the modified LLM under the same name and with the same element version number only if no other task is currently processing this LLM.

## 4.2 Outputs from BINDER

BINDER outputs are as follows:

- saved LLMs. Each LLM is saved with its own element name in a program library.
- lists containing status information relating to the current LLM or concluding information about the saved LLM.

With the aid of the display functions (see [section “Display functions” on page 126ff](#)), the user can display information about default values, symbols and library elements.

### 4.2.1 Saved LLMs

BINDER can also link *multiple* LLMs in a single BINDER run. Each SAVE-LLM statement saves the current LLM with its current structure as a *separate* type L library element in a program library. LLMs can only be saved by means of the SAVE-LLM statement. If a new START-LLM-CREATION or START-LLM-UPDATE statement is issued without the current LLM having been previously saved by means of the SAVE-LLM statement, the previous current LLM will be overwritten. The END statement terminates the BINDER run without saving the current LLM.

### 4.2.2 Lists

Lists containing information about the status of the LLM are output by means of the SHOW-MAP statement, lists containing concluding information about the saved LLM are output with the SAVE-LLM statement (MAP=YES operand).

The output destination may be:

- the system file SYSLST.
- a file that is output to SYSOUT automatically by means of an implicit SHOW-FILE command (see the “Commands” manual [5]). This file is created under the default file name and deleted immediately after it has been output unless the user prevents this.
- a file whose name is specified by the user.
- a file defined by the default file link name or a specified file link name.
- a user-written subroutine which evaluates the supplied information. This information is output to an ISAM file which the subroutine must access.

These output destinations are also available in the SHOW-LIBRARY-ELEMENTS statement.

The lists generated by SHOW-MAP or SHOW-LIBRARY-ELEMENTS (except for those output on SYSLST) are by default ISAM files with ISAM keys with a length of 8. The ISAM key can be used for evaluation of the lists. A description of the ISAM keys is provided in the appendix ([page 369f](#)).

The operands in the SHOW-MAP statement determine which lists of information are output. Default settings can be modified by a preceding MODIFY-MAP-DEFAULTS statement. A list with the name \*STD is always generated automatically. The user can, however, also define other lists in a BINDER run with the MODIFY-MAP-DEFAULTS statement and can name them individually (MAP-NAME operand). The user can then, for example, define different default values for each list and/or determine the scope of the lists. If the list specified in MAP-NAME does not already exist, it is created; otherwise, the values for the list are simply updated. SHOW-MAP *cannot* be used to define lists; this can be done only with the MODIFY-MAP-DEFAULTS statement. A SHOW-MAP statement for an existing list has no effect on subsequent SHOW-MAP statements and changes nothing in the other MAP definitions.

The following table provides an overview of the various lists with their associated operands. The individual lists are then illustrated by way of an example.

A header line containing a user comment can be defined for all lists (USER-COMMENT operand).

List	Operands
Help information (list of abbreviations)	HELP-INFORMATION
Global information	GLOBAL-INFORMATION
Logical structure	LOGICAL-STRUCTURE
Physical structure	PHYSICAL-STRUCTURE
Program map	PROGRAM-MAP
Unresolved definitions list (list of unresolved external references)	UNRESOLVED-LIST
Sorted symbols definitions list (sorted list of program definitions)	SORTED-PROGRAM-MAP
Pseudo-registers list	PSEUDO-REGISTER
Unused modules list	UNUSED-MODULE-LIST
Merged modules (shown in other lists)	MERGED-MODULES
Duplicate symbols definitions list (list of multiple program definitions)	DUPLICATE-LIST
Input information	INPUT-INFORMATION
Statement list	STATEMENT-LIST

**Example of list output**

```

/start-binder _____ (1)
//mod-map-def user-comment=c'LIST EXAMPLE', - _____ (2)
//          help-information=*yes, - _____ (3)
//          global-information=*yes, - _____ (4)
//          logical-structure=*yes(res-scope=*yes,hsi-code=*yes), - _____ (5)
//          physical-structure=*yes, - _____ (6)
//          program-map=*par(def=*all,inv-xref-list=*all,ref=*all), - _____ (7)
//          unresolved-list=*yes( - _____ (8)
//                                noref=*yes), - _____ (9)
//          sorted-program-map=*yes, - _____ (10)
//          pseudo-register=*yes, - _____ (11)
//          unused-module-list=*yes, - _____ (12)
//          duplicate-list=*yes(inverted-xref-list=*yes), - _____ (13)
//          merged-modules=*yes, - _____ (14)
//          input-information=*yes, - _____ (15)
//          statement-list=*yes, - _____ (16)
//          output=*syslst _____ (17)
//start-statement-recording _____ (18)
//start-llm-creation internal-name=complex1 _____ (19)
//include-modules library=bnd.llmlib,element=(autolinkl,autolinkr) _____ (20)
//save-llm library=bnd.llmlib,element=complex1 _____ (21)
//end _____ (22)

```

- (1) BINDER call
- (2) Definition of defaults for the BINDER lists. A header line with the title “LIST EXAMPLE” is output for all lists.
- (3) Help information
- (4) Global information
- (5) Logical structure list
- (6) Physical structure list
- (7) Program map
- (8) Unresolved definitions list
- (9) Not referenced symbols list
- (10) Sorted symbols definition list
- (11) Pseudo-registers list
- (12) Unused modules list
- (13) Duplicate symbols definitions list
- (14) Merged modules are also output in the lists

- (15) Input information
- (16) Recorded BINDER statements list
- (17) The output destination is system file SYSLST
- (18) Switching on the recording of BINDER statements
- (19) An LLM with the internal name COMPLEX1 is generated
- (20) LLMs AUTOLINKL and AUTOLINKR, which are stored in program library BND.LLMLIB, are included in the LLM
- (21) The LLM is stored in program library BND.LLMLIB as a type L element. The library element contains the element name COMPLEX1. The BINDER statement SAVE-LLM initiates output of the BINDER list provided MAP=YES is specified in SAVE-LLM. The defaults set in (2) to (17) are used for list output.
- (22) Termination of the BINDER run

The lists output under points (3) through (16) are described in detail below.

### Help information (3)

The abbreviations used in the subsequent lists are explained in this list (e.g. SD for CSECT).

BINDER V02.7A \*HELP INFORMATION\* LLM: COMPLEX1 DATE=2016-02-03 10:43:51 PAGE 1

LIST EXAMPLE  
ATTRIBUTES CONVERSION

```

----- AMODE: S=24 BITS, X=31 BITS, A=ANY
| +----- RMODE: S=24 BITS, A=ANY
| | +----- ALIGNMENT: P=PAGE ALIGNED, O=OTHER
| | | +----- ACCESS: R=READ-ONLY
| | | | +----- INVISIBILITY: I=INVISIBLE, R=RUN-TIME
| | | | | +----- RESIDENT: R=RESIDENT
| | | | | | +----- PUBLIC/PRIVATE: P=PUBLIC
| | | | | | | +----- PRIVILEGE: P=PRIVILEGED
| | | | | | | |
V V V V V V V V V V V V ( . = NOT SPECIFIED)
S S . R I R P P
X A P . R . . .
A O

```

MAP GLOSSARY

```

-----
: CM : COMMON                : SD : CSECT                    :
: ER : EXTERN                : SUB: SUB-LLM               :
: GM : GROSSMODULE          : VC : VCON                    :
: LD : ENTRY                : WX : WEAK-EXTERN           :
: LLM: LINK AND LOAD MODULE : XD : XDSECT-DEF              :
: OM : OBJECT MODULE        : XR : XDSECT-REF              :
: RT : RUN-TIME            :   :                             :
-----

```

--- END OF SECTION ---

GROSSMODULE = prelinked module



*Global information (4)*

This list contains the following information:

**ENTRY POINT**

Address and position of the start address of the LLM (relative to the associated slice), HSI code, memory access mode and name of the symbol.

This information is significant only after the LLM has been saved. Currently the memory access mode only applies to internal applications on systems with SPARC architecture.

**LOAD POINT**

Load address of the LLM.

**LLM LENGTH**

Length of the LLM.

**COPYRIGHT**

Copyright information.

The following information is significant only after the LLM has been saved.

**LIBRARY NAME**

File name of the program library in which the LLM has been saved.

**ELEMENT NAME/VERSION**

Element name and element version of the LLM in the program library.

**LLM FORMAT**

The format in which the LLM was stored wurde

Format 1: It is possible to load the LLM in all BS2000 versions.

Format 2: It is possible to load the LLM in all BS2000 versions.

Format 3: It is only possible to load the LLM as of BS2000 V3.0A.

Format 4: It is only possible to load the LLM with BLSSERV as of BS2000 V3.0A.

**SLICE TYPE**

Type of slice:

**SINGLE** Single slice

**BY-USER** User-defined slice

**BY-ATTR** Slice by attributes

In addition, those attributes from which the slice was formed are output. Value abbreviated as per list of abbreviations in help information (see [page 144](#)).

SYMBOL DICTIONARY

External Symbols Vector stored: yes/no

RELOCATION DATA

Relocation data stored: yes/no

LOGICAL STRUCTURE

Type of structure information stored

WHOLE-LLM Entire structure information

OBJECT-MODULES Only one structure with the internal name of the LLM and the object modules is stored

NONE Only the internal name of the LLM is stored

TEST SUPPORT

List for symbolic debugging stored: yes/no



This value only shows the setting of the TEST-SUPPORT operand of the relevant BINDER statement. Information on the existence of the list for symbolic debugging is provided in the "T&D" column in BINDER lists.

```

BINDER V02.7A  *GLOBAL INFORMATION*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE    2
LIST EXAMPLE
      ADDRESS  IN SLICE          HSI  MMODE  SYMBOL
-----
ENTRY POINT: 00000000      1 + 00000000 /7500  TU4K  AUTOA
LOAD POINT:  00000000
-----
LLM LENGTH:  00000800
-----
COPYRIGHT: 2016 FUJITSU TECHNOLOGY SOLUTIONS GMBH. ALL RIGHTS RESERVED
-----
LIBRARY NAME: :CTID:$USERID.BND.LLMLIB
-----
ELEMENT NAME/VERSION: COMPLEX1/@
-----
LLM FORMAT:      1
-----
SLICE TYPE: SINGLE
-----
SYMBOL DICTIONARY: YES
-----
RELOCATION DATA: YES
-----
LOGICAL STRUCTURE: WHOLE-LLM
-----
TEST SUPPORT: YES
-----

```

--- END OF SECTION ---

*Logical structure (5)*

This list comprises two parts.

The first one is a representation of the structure tree of the LLM. The list contains the internal names of the included object modules and sub-LLMs in the order in which they were included in the structure tree (from left to right).

Meaning:

SLICE	Identifier for the slice Refers to the SLICE column in the physical structure list (see <a href="#">page 149</a> ). If a logical node is spread over more than one slice, '-' must be entered for the number of the slice.														
TYPE	Node type in the structure tree of the LLM. <table> <tr> <td>LLM</td> <td>Root</td> </tr> <tr> <td>OM</td> <td>Object module</td> </tr> <tr> <td>GM</td> <td>Prelinked module</td> </tr> <tr> <td>SUB</td> <td>Sub-LLM</td> </tr> <tr> <td>RT</td> <td>Runtime (if the module is to be regarded as a runtime module)</td> </tr> </table>	LLM	Root	OM	Object module	GM	Prelinked module	SUB	Sub-LLM	RT	Runtime (if the module is to be regarded as a runtime module)				
LLM	Root														
OM	Object module														
GM	Prelinked module														
SUB	Sub-LLM														
RT	Runtime (if the module is to be regarded as a runtime module)														
HSI	Type of code of the LLM. <table> <tr> <td>/7500</td> <td>/390 code</td> </tr> <tr> <td>/4000</td> <td>RISC(MIPS) code</td> </tr> <tr> <td>MIXED</td> <td>Mixed binary code (= /390 + RISC(MIPS))</td> </tr> <tr> <td>SPARC</td> <td>SPARC code</td> </tr> <tr> <td>MIXSP</td> <td>Mixed SPARC code (= /390 + SPARC)</td> </tr> <tr> <td>X86</td> <td>X86 code</td> </tr> <tr> <td>MIXX8</td> <td>Mixed X86 code (= /390 + X86)</td> </tr> </table>	/7500	/390 code	/4000	RISC(MIPS) code	MIXED	Mixed binary code (= /390 + RISC(MIPS))	SPARC	SPARC code	MIXSP	Mixed SPARC code (= /390 + SPARC)	X86	X86 code	MIXX8	Mixed X86 code (= /390 + X86)
/7500	/390 code														
/4000	RISC(MIPS) code														
MIXED	Mixed binary code (= /390 + RISC(MIPS))														
SPARC	SPARC code														
MIXSP	Mixed SPARC code (= /390 + SPARC)														
X86	X86 code														
MIXX8	Mixed X86 code (= /390 + X86)														
MMODE	Memory access mode of the LLM. <table> <tr> <td>TU4K</td> <td>TU code 4K dependent</td> </tr> <tr> <td>NATIV</td> <td>NATIVE TU code independent of the memory page size</td> </tr> <tr> <td>COMP</td> <td>COMPATIBLE code linkable with TU4K or NATIV</td> </tr> <tr> <td>TPR</td> <td>TPR code</td> </tr> <tr> <td>UNDEF</td> <td>Undefined MMODE</td> </tr> <tr> <td>UNKNW</td> <td>Unknown MMODE</td> </tr> <tr> <td>UNSP</td> <td>Unspecified MMODE</td> </tr> </table>	TU4K	TU code 4K dependent	NATIV	NATIVE TU code independent of the memory page size	COMP	COMPATIBLE code linkable with TU4K or NATIV	TPR	TPR code	UNDEF	Undefined MMODE	UNKNW	Unknown MMODE	UNSP	Unspecified MMODE
TU4K	TU code 4K dependent														
NATIV	NATIVE TU code independent of the memory page size														
COMP	COMPATIBLE code linkable with TU4K or NATIV														
TPR	TPR code														
UNDEF	Undefined MMODE														
UNKNW	Unknown MMODE														
UNSP	Unspecified MMODE														

LEVEL Node level in the structure tree of the LLM.  
 STR# Progressive number of the node in the LLM structure tree  
 This number is referred to in the second part of the list.  
 NAME Internal name of the node in the structure tree of the LLM.  
 T&D List for symbolic debugging present: yes/no

```

BINDER V02.7A *LOGICAL STRUCTURE* LLM: COMPLEX1 DATE=2016-02-03 10:43:51 PAGE 3
LIST EXAMPLE
SLICE TYPE HSI MMODE LEVEL STR# NAME T&D
-----
1 LLM /7500 TU4K 0 1 COMPLEX1
1 SUB /7500 TU4K 1 2 AUTOLINKL
1 SUB /7500 TU4K 2 3 SUB1
1 OM /7500 TU4K 3 4 AUTOA NO
1 OM /7500 TU4K 3 5 AUTOC NO
1 OM /7500 TU4K 3 6 AUTO2 NO
1 SUB /7500 TU4K 2 7 SUB2
1 OM /7500 TU4K 3 8 AUTOA NO
1 SUB /7500 TU4K 3 9 SUB3
1 OM /7500 TU4K 4 10 AUTOA NO
1 OM /7500 TU4K 4 11 AUTOC NO
1 OM /7500 TU4K 4 12 AUTO2 NO
1 SUB /7500 TU4K 1 13 AUTOLINKR
1 SUB /7500 TU4K 2 14 SUB1
1 OM /7500 TU4K 3 15 AUTO21 NO
1 SUB /7500 TU4K 2 16 SUB2
1 OM /7500 TU4K 3 17 AUTO22 NO
1 SUB /7500 TU4K 3 18 SUB3
1 OM /7500 TU4K 4 19 AUTO23 NO
    
```

--- END OF SECTION ---

The second part of the list shows the path names specified in operands HIGH-PRIORITY-SCOPE, LOW-PRIORITY-SCOPE or FORBIDDEN-SCOPE.

PATH Progressive number of the path  
 This number is referred to in the first part of the list.

STR# Progressive number of the node in the LLM structure tree  
 This number is used to refer to the first part of the list.

PATHNAME Path name specified in operand HIGH-PRIORITY-SCOPE, LOW-PRIORITY-SCOPE or FORBIDDEN-SCOPE.

```

BINDER V02.7A *SCOPE PATH INFORMATION* LLM: COMPLEX1 DATE=2016-02-03 10:43:51 PAGE 4
LIST EXAMPLE
PATH STR# PATHNAME
-----
    
```

--- END OF SECTION ---

*Physical structure (6)*

This list contains the following information:

- SLICE** Identifier for the slice or region.  
This identifier is referenced in the SLICE column in other lists.
- ATTRIB** Attributes of the slice.  
Value abbreviated as per list of abbreviations (see [page 144](#)).
- LENGTH** Length of the slice.  
The length refers to the length of the text information (TXT). All slices having the length 0 are also logged in the list. Slices with length 0 are, however, only taken over when the LLM is saved if they have been defined by the user. Single slices with length 0 and slices by attributes with length 0 are not saved in the LLM.
- NAME** Name of the slice or region (see [page 77](#)).

```

BINDER V02.7A  *PHYSICAL STRUCTURE*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE    5
LIST EXAMPLE
SLICE  ATTRIB  LENGTH  NAME
-----
1 .A..... 00000800 PUU-ROU-RTU-RMU

```

--- END OF SECTION ---

*Program map (7)*

The program map describes the modules that constitute the LLM in terms of their name, load address, length, attributes and origin. The program map may optionally contain:

- the program definitions and COMMONs in the individual modules (DEFINITIONS operand)
- the references in the individual modules (REFERENCES operand)
- a cross-reference list (INVERTED-XREF-LIST operand)  
The latter describes the list of external references that are resolved by a current program definition (CSECT, ENTRY, ..).

The program map contains the following individual items of information:

OBJ	Type of the module or symbol. Value abbreviated as per list of abbreviations (see <a href="#">page 144</a> ).
NAME	Name of the module or symbol
ADDRESS	Load address for CSECTs and ENTRYs
LENGTH	Length of the text information (TXT) for CSECTs
ATTRIB	Attributes of the CSECTs. Value abbreviated as per list of abbreviations (see <a href="#">page 144</a> ). For information purposes, also specified for ENTRYs (only the attribute INVISIBLE is relevant).
SLICE	Identifier for the slice. Refers to the SLICE column in the physical structure list (see <a href="#">page 149</a> ).
TYPE	Of significance only to the cross-reference list (INVERTED-XREF-LIST). Specifies the type of resolved references. Value abbreviated as per list of abbreviations (see <a href="#">page 144</a> ).
RESOLVED	Of significance only to references. Address that is entered in the resolved references. If an address is entered that was defined with the SET-EXTERN-RESOLUTION statement, this column will contain the entry EXT-RES. In this case, the name of the reference and the entered address are logged in an additional line.
IN MODULE	Of significance only to references and to the cross-reference list (INVERTED-XREF-LIST). Specifies the module in which the external references specified under OBJ and NAME were resolved.

FROM ELEMENT#	<p>Of significance only to object modules (OMs). Contains the identifier for the element. This identifier is defined in the ELEM# column of the input information list (see <a href="#">page 156</a>).</p>																
OM#	<p>Identifier for the object module (OM). This identifier is referenced in the other lists by being prefixed to the module name.</p>																
SLICE	<p>Of significance only to references and the cross-reference list (INVERTED-XREF-LIST). Specifies the slice containing the symbol that resolves the external reference specified under OBJ. Refers to the SLICE column in the physical structure list (see <a href="#">page 149</a>).</p>																
SATIS	<p>Specifies how external references were handled in the LLM.</p> <table><tr><td>NOREF</td><td>The external reference was not resolved because no relocation dictionary (RLD) was available.</td></tr><tr><td>UNRES</td><td>Unresolved external reference.</td></tr><tr><td>SLICE</td><td>The external reference was resolved in this slice.</td></tr><tr><td>ERREX</td><td>An external reference for which an address was entered with the SET-EXTERN-RESOLUTION statement is present.</td></tr></table> <p>Additional values for user-defined slices:</p> <table><tr><td>HIGH</td><td>The symbol was resolved in a slice that is higher than the slice containing the symbol.</td></tr><tr><td>LOW</td><td>The symbol was resolved in a slice that is lower than the slice containing the symbol.</td></tr><tr><td>OTHER</td><td>The symbol was resolved in a slice that lies in a different region to the slice containing the symbol.</td></tr><tr><td>CONC</td><td>The symbol was resolved in a slice that is exclusive with respect to the slice containing the symbol.</td></tr></table>	NOREF	The external reference was not resolved because no relocation dictionary (RLD) was available.	UNRES	Unresolved external reference.	SLICE	The external reference was resolved in this slice.	ERREX	An external reference for which an address was entered with the SET-EXTERN-RESOLUTION statement is present.	HIGH	The symbol was resolved in a slice that is higher than the slice containing the symbol.	LOW	The symbol was resolved in a slice that is lower than the slice containing the symbol.	OTHER	The symbol was resolved in a slice that lies in a different region to the slice containing the symbol.	CONC	The symbol was resolved in a slice that is exclusive with respect to the slice containing the symbol.
NOREF	The external reference was not resolved because no relocation dictionary (RLD) was available.																
UNRES	Unresolved external reference.																
SLICE	The external reference was resolved in this slice.																
ERREX	An external reference for which an address was entered with the SET-EXTERN-RESOLUTION statement is present.																
HIGH	The symbol was resolved in a slice that is higher than the slice containing the symbol.																
LOW	The symbol was resolved in a slice that is lower than the slice containing the symbol.																
OTHER	The symbol was resolved in a slice that lies in a different region to the slice containing the symbol.																
CONC	The symbol was resolved in a slice that is exclusive with respect to the slice containing the symbol.																

BINDER V02.7A \*PROGRAM MAP\* LLM: COMPLEX1 DATE=2016-02-03 10:43:51 PAGE 6

LIST EXAMPLE

OBJ	NAME	ADDRESS	LENGTH	ATTRIB	SLICE	TYPE	RESOLVED	IN	MODULE/FROM	ELEMENT#	OM#	SLICE	SATIS
OM	AUTOA				1						1		
SD	AUTOA	00000000	00000040	AA.....	1								
ER	AUTOB						FFFFFFFF						UNRES
WX	AUTOWX						FFFFFFFF						UNRES
VC	ENTRYB1						FFFFFFFF						UNRES
OM	AUTOA				1						2		
SD	AUTOA	00000040	00000018	AA.....	1								
VC	AUTOCOM						FFFFFFFF						UNRES
OM	AUTO2				1						3		
SD	AUTO2	00000058	0000000C	AA.....	1								
VC	AUTO21						00000110	AUTO21			8	1	SLICE
VC	AUTO22						00000118	AUTO22			9	1	SLICE
VC	AUTO23						00000168	AUTO23			10	1	SLICE
OM	AUTOA				1						4		
SD	AUTOA	00000068	00000040	AA.....	1								
ER	AUTOB						FFFFFFFF						UNRES
WX	AUTOWX						FFFFFFFF						UNRES
VC	ENTRYB1						FFFFFFFF						UNRES
OM	AUTOA				1						5		
SD	AUTOA	000000A8	00000040	AA.....	1								
ER	AUTOB						FFFFFFFF						UNRES
WX	AUTOWX						FFFFFFFF						UNRES
VC	ENTRYB1						FFFFFFFF						UNRES
OM	AUTOA				1						6		
SD	AUTOA	000000E8	00000018	AA.....	1								
VC	AUTOCOM						FFFFFFFF						UNRES
OM	AUTO2				1						7		
SD	AUTO2	00000100	0000000C	AA.....	1								
VC	AUTO21						00000110	AUTO21			8	1	SLICE
VC	AUTO22						00000118	AUTO22			9	1	SLICE
VC	AUTO23						00000168	AUTO23			10	1	SLICE
OM	AUTO21				1						8		
SD	AUTO21	00000110	00000006	AA.....	1								
						VC		AUTO2			3	1	SLICE
						VC		AUTO2			7	1	SLICE
OM	AUTO22				1						9		
SD	AUTO22	00000118	00000006	AA.....	1								
						VC		AUTO2			3	1	SLICE
						VC		AUTO2			7	1	SLICE
OM	AUTO23				1						10		
SD	AUTO23	00000168	00000006	AA.....	1								
						VC		AUTO2			3	1	SLICE
						VC		AUTO2			7	1	SLICE

--- END OF SECTION ---



*COMMON list*

This list contains the following information:

NAME Name of the COMMON

## ADDRESS

Address of the CSECT with which the COMMON was promoted; FFFFFFFF if the COMMON was not promoted.

LENGTH Length of the COMMON

ATTRIB Attributes of the COMMON.

Value abbreviated as per list of abbreviations (see [page 144](#)).

TYPE Of significance only to the cross-reference list (INVERTED-XREF-LIST).

Specifies the type of the resolved references. Value abbreviated as per list of abbreviations (see [page 144](#)).

## IN MODULE

Name of the module containing the COMMON.

OM# This identifier is referred to in other lists by placing the identifier in front of the symbol.

SLICE Of significance only to the cross-reference list (INVERTED-XREF-LIST).

Identifier for the slice. Refers to the SLICE column in the physical structure list (see [page 149](#)).

```

BINDER V02.7A  *COMMON LIST*                LLM:  COMPLEX1    DATE=2016-02-03 10:43:51 PAGE   7
LIST EXAMPLE
NAME          ADDRESS  LENGTH  ATTRIB  TYPE  IN MODULE          OM#  SLICE  SATIS
-----
AUTOC2        FFFFFFFF 00000050 AA..... AUTOC                2    1
                *NOT-PROMOTED
AUTOC2        FFFFFFFF 00000050 AA..... AUTOC                6    1
                *NOT-PROMOTED
AUTO22        00000118 00000050 AA..... AUTO23              10    1
                AUTO22                9    1
                --- END OF SECTION ---

```

*Unresolved definitions list (8)*

This list contains the names of unresolved external references for which the address X'FFFFFFFF' is entered in the RESOLVED column and the value UNRES is entered in the SATIS column in the program map (see [page 151](#)).

```

BINDER V02.7A  *UNRESOLVED REFERENCES*          LLM:  COMPLEX1    DATE=2016-02-03 10:43:51 PAGE   8
LIST EXAMPLE
VC AUTOCOM    VC ENTRYB1    WX AUTOWX    ER AUTOB
                --- END OF SECTION ---

```

*List of the unreferenced external references (9)*

This list contains the names of the unreferenced external references for which the value NOREF is entered in the SATIS column of the program overview (see [page 151](#)).

```
BINDER V02.7A *NOT REFERENCED SYMBOLS*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE    9
LIST EXAMPLE
NONE
                                         --- END OF SECTION ---
```

*Sorted symbol definitions list (10)*

This list contains the following information:

- NAME      Name of the program definition
- TYPE      Specifies the type of program definition.  
Value abbreviated as per list of abbreviations (see [page 144](#)).
- IN MODULE      Name of the module containing the program definition
- OM#      Identifier for the object module containing the symbol. Refers to column OM# in the program map (7).
- SLICE      Specifies the slice containing the program definition. Refers to the SLICE column in the physical structure list (see [page 149](#)).

```
BINDER V02.7A *SORTED SYMBOLS*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE   10
LIST EXAMPLE
NAME                                     TYPE IN MODULE          OM#  SLICE
----                                     -
AUTOA                                     SD  AUTOA                1    1
AUTOA                                     SD  AUTOA                4    1
AUTOA                                     SD  AUTOA                5    1
AUTOA                                     SD  AUTOA                5    1
AUTOC                                     SD  AUTOC                2    1
AUTOC                                     SD  AUTOC                6    1
AUTOC2                                    CM  AUTOC                2    1
AUTOC2                                    CM  AUTOC                6    1
AUTO2                                     SD  AUTO2                3    1
AUTO2                                     SD  AUTO2                7    1
AUTO21                                    SD  AUTO21               8    1
AUTO22                                    SD  AUTO22               9    1
AUTO22                                    CM  AUTO23              10    1
AUTO23                                    SD  AUTO23              10    1
                                         --- END OF SECTION ---
```

*Pseudo-registers list (11)*

This list contains the following information:

NAME      Name of the pseudo-register  
 OFFSET    Reference point for the pseudo-register  
 LENGTH    Length assigned to the pseudo-register  
 REF LEN   Length of the pseudo-register used by the module  
 IN MODULE  
           Name of the module containing the pseudo-register

Last line: PSEUDO-REGISTER VECTOR LENGTH  
 Specifies the length of the pseudo-register vector.

```
BINDER V02.7A  *PSEUDO-REGISTERS*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE  11
LIST EXAMPLE
PSEUDO-REGISTERS  NAME                      OFFSET  LENGTH  REF LEN  IN MODULE
-----
PSEUDO-REGISTER VECTOR LENGTH:           0
-----
--- END OF SECTION ---
```

*Unused modules list (12)*

This list contains the names of those modules in which no symbols were used for resolving external references.

The number specifies the module identifier defined for the module in the program map in the OM# column (see [page 151](#)).

```
BINDER V02.7A  *UNUSED MODULES*          LLM:  COMPLEX1      DATE=2016-02-03 10:43:51 PAGE  12
LIST EXAMPLE
UNUSED MODULES LIST
-----
 1 AUTOA                2 AUTOC                3 AUTO2
 4 AUTOA                5 AUTOA                6 AUTOC
 7 AUTO2
-----
--- END OF SECTION ---
```

*Duplicate symbol definitions list (13)*

The entries have the same meaning as in the program map (see [page 150](#)).

```

BINDER V02.7A  *DUPLICATE SYMBOLS*          LLM: COMPLEX1    DATE=2016-02-03 10:43:51 PAGE  13
LIST EXAMPLE
NAME           TYPE IN MODULE              OM#  SLICE TYPE IN MODULE              OM#  SLICE
-----
AUTOA          SD  AUTOA                1    1
AUTOA          SD  AUTOA                4    1
AUTOA          SD  AUTOA                5    1
AUTOA          SD  AUTOA                2    1
AUTOA          SD  AUTOA                6    1
AUTOA          CM  AUTOA                2    1
AUTOA          CM  AUTOA                6    1
AUTOA          SD  AUTO2                3    1
AUTOA          SD  AUTO2                7    1
AUTOA          SD  AUTO22               9    1
AUTOA          VC  AUTO2                3    1
AUTOA          VC  AUTO2                7    1
AUTO22        CM  AUTO23               10    1
    
```

--- END OF SECTION ---

*Merged modules list (14)*

Merged modules (if any) appear in all lists containing symbols. The merged modules are indicated by parentheses. The resultant module is a prelinked module that is marked with GM in the lists. An example is provided on [page 159ff](#).

*Input information (15)*

The list comprises two parts:

First part of list

The list contains the following information for each included module:

- NAME      Element name
- TYPE      Element type in the program library
  - L            LLM
  - R            Object module (OM)
- VERSION   Element version in the program library. The entry @ signifies the default value for the highest version for program libraries (see the "LMS" manual [3]).
- DATE      Creation date
- FILE ID   Identifier for the program library or program file. This provides the link with the second part of the list.

- BIND** Specifies how the module (ELEMENT) was included.
  - EXPL** The module was included by means of the INCLUDE-MODULES or REPLACE-MODULES statement.
  - AUTO** The module was included by means of autolink.
  - IMPL** The module was included through an INCLUDE record in the object module.
- ELEM#** Identifier for the element.  
This identifier is referenced in the other lists.
- PATH** Path name of the sub-LLM branched to in the INCLUDE-MODULES or REPLACE-MODULES statement. This is followed by a list containing the names of all object modules (OMs) comprising the sub-LLM (see logical structure list, [page 147](#)).  
Each module name is prefixed by a number for the module, defined for the module in the program map in the OM# column (see [page 151](#)).

```

BINDER V02.7A *INPUT INFORMATION* LLM: COMPLEX1 DATE=2016-02-03 10:43:51 PAGE 14
LIST EXAMPLE
NAME TYPE VERSION DATE FILE ID BIND ELEM#
-----
ELEMENT: AUTOLINKL L @ 1991-02-01 1 EXPL 1
-----
PATH: AUTOLINKL
MODULES
-----
1 AUTOA 2 AUTOC 3 AUTO2
4 AUTOA 5 AUTOA 6 AUTOC
7 AUTO2
ELEMENT: AUTOLINKR L @ 1992-10-02 1 EXPL 2
-----
PATH: AUTOLINKR
MODULES
-----
8 AUTO21 9 AUTO22 10 AUTO23
--- END OF SECTION ---
    
```

**Second part of the list**

The list contains the following information for each input source:

**FILE ID** Identifier for the input source.  
Provides the link with the first part of the list.

**LINKNAME**  
File link name of the input source

**FILE TYPE**  
Type of the input source

- PLAM** Program library (element type R or L)
- OML** Object module library

OMF EAM object module file

## FILENAME

File name of the input source

```

BINDER V02.7A *LINKNAME CONVERSION*          LLM: COMPLEX1    DATE=2016-02-03 10:43:51 PAGE 15
LIST EXAMPLE
FILE ID  LINKNAME  FILE TYPE  FILENAME
-----  -
1          PLAM      :CTID:$USERID.BND.LLMLIB

```

--- END OF SECTION ---

*List of the BINDER statements (16)*

This list contains the list of the recorded BINDER statements (see //START-STATEMENT-RECORDING).

```

BINDER V02.7A *BINDER STATEMENTS*          LLM: COMPLEX1    DATE=2016-02-03 10:43:51 PAGE 17
LIST EXAMPLE
START-LLM-CREATION INTERNAL-NAME=COMPLEX1,COPYRIGHT=*PARAMETERS,INCLUSION-DEFAULTS=*PARAMETERS
INCLUDE-MODULES MODULE-CONTAINER=*LIBRARY-ELEMENT(LIBRARY=BNDBSP.LIB,ELEMENT=(AUTOLINKL,AUTOLINKR),
TYPE>(*L,*R))
SAVE-LLM MODULE-CONTAINER=*LIBRARY-ELEMENT(LIBRARY=#OUT.PL,ELEMENT=COMPLEX1),MAP=*YES

```

--- END OF SECTION ---

### List example for an LLM with merged modules

The example below shows how BINDER lists appear when merged modules are present in the LLM.

```

/start-binder
% BND0500 BINDER VERSION 'V02.7A00' STARTED
//modify-map-defaults help-information=no,-
//                                global-information=no,-
//                                logical-structure=yes,-
//                                physical-structure=no,-
//                                program-map=parameters(definitions=all,-
//                                inverted-xref-list=all,references=all),-
//                                unresolved-list=yes,-
//                                sorted-program-map=yes,-
//                                pseudo-register=no,-
//                                unused-module-list=no,-
//                                duplicate-list=yes(inv-xref-list=yes),-
//                                input-information=no,-
//                                output=*syslst _____ (1)
//start-llm-creation internal-name=complex2 _____ (2)
//begin-sub-llm-statements sub-llm-name=subllm _____ (3)
//include-modules library=bnd.llmlib,element=(auto2,auto21)
//end-sub-llm-statements
% BND1120 CURRENT LOGICAL POSITION: 'COMPLEX2'
//include-modules library=bnd.llmlib,element=auto22 _____ (4)
//show-map user-comment='LIST EXAMPLE PRIOR TO MERGE',MERGED-MODULES=YES (5)
//merge-modules name=subllm,path-name=complex2 _____ (6)
% BND1112 '0' KEPT ENTRIES
//show-map user-comment='LIST EXAMPLE (MERGED-MODULES=YES)',- _____ (7)
//                                merged-modules=yes
//show-map user-comment='LIST EXAMPLE (MERGED-MODULES=NO)',- _____ (8)
//                                merged-modules=no
//save-llm library=bnd.llmlib,element=complex2,map=no _____ (9)
% BND3101 SOME EXTERNAL REFERENCES ARE UNRESOLVED
% BND1501 LLM FORMAT : '1'
//end
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'UNRESOLVED EXTERNAL'

```

- (1) The default values for the BINDER lists were defined after BINDER was started. Only those modules are to be output that contain information relevant for merging modules.
- (2) An LLM with the internal name COMPLEX2 is generated.
- (3) LLM COMPLEX2 contains a sub-LLM with the name SUBLLM. Object modules AUTO2 and AUTO21 are included in this. The two object modules are fetched from program library BND.LLMLIB. The sub-LLM is terminated.

- (4) A further object module (AUTO22) is included in the LLM.
- (5) The BINDER lists are displayed prior to the merge procedure. The lists contain the comment 'LIST EXAMPLE PRIOR TO MERGE' in the header. MERGED-MODULES=YES causes any merged modules already contained in the lists to be displayed in parentheses. This is not the case in LLM COMPLEX2, however, as can be seen in the lists.
- (6) Sub-LLM SUBLLM is merged. It is unambiguously defined as an object for merging via the path COMPLEX2 and its name SUBLLM.
- (7) The BINDER lists are output containing the merged modules. The lists contain the comment 'LIST EXAMPLE (MERGED-MODULES=YES)' in the header. The merged modules are marked in the lists by parentheses. A prelinked module with the name SUBLLM has been created.
- (8) The BINDER lists are output without the merged modules. The lists contain the comment 'LIST EXAMPLE (MERGED-MODULES=NO)' in the header. In the lists the only indication of the merge procedure is that the prelinked module created, SUBLLM, also appears as a module in the program map and is marked with the abbreviation GM.
- (9) LLM COMPLEX2 is stored in program library BND.LLMLIB as a type L element under the name COMPLEX2. No further list is output. BINDER recognizes that not all external references have been resolved and that the LLM will be stored in format 1 (see [page 44](#)).



The following lists are output to SYSLST. The lists are identified by their header lines. The parentheses in the lists indicate merged modules.

*Lists prior to the merge run*

```

BINDER V02.7A  *LOGICAL STRUCTURE*          LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   1
LIST EXAMPLE PRIOR TO MERGE
SLICE TYPE  HSI   MMODE LEVEL STR#  NAME                                     T&D
-----
1 LLM  /7500  TU4K    0     1  COMPLEX2
1 SUB  /7500  TU4K    1     2  SUBLLM
1 OM   /7500  TU4K    2     3  AUTO2
1 OM   /7500  TU4K    2     4  AUTO21
1 OM   /7500  TU4K    1     5  AUTO22
                                                    NO
                                                    NO
                                                    NO
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *SCOPE PATH INFORMATION*      LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   2
LIST EXAMPLE PRIOR TO MERGE
PATH  STR#  PATHNAME
-----
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *PROGRAM MAP*                LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   3
LIST EXAMPLE PRIOR TO MERGE
OBJ  NAME      ADDRESS      LENGTH      ATTRIB  SLICE TYPE RESOLVED IN MODULE/FROM ELEMENT#  OM#  SLICE  SATIS
-----
OM   AUTO2                1
SD   AUTO2  00000000  00000000  AA.....  1
VC   AUTO21                2      1 SLICE
VC   AUTO22                3      1 SLICE
VC   AUTO23                FFFFFFFF UNRES
OM   AUTO21                2
SD   AUTO21  00000010  00000006  AA.....  1
VC   AUTO21                1      1 SLICE
OM   AUTO22                3
SD   AUTO22  00000018  00000006  AA.....  1
VC   AUTO22                1      1 SLICE
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *COMMON LIST*                LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   4
LIST EXAMPLE PRIOR TO MERGE
NAME      ADDRESS      LENGTH      ATTRIB  TYPE  IN MODULE      OM#  SLICE  SATIS
-----
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *UNRESOLVED REFERENCES*      LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   5
LIST EXAMPLE PRIOR TO MERGE
VC AUTO23
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *UNRESOLVED LONG NAMES*      LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   6
LIST EXAMPLE PRIOR TO MERGE
INDEX NAME
-----
                                                    --- END OF SECTION ---
    
```

```

BINDER V02.7A  *SORTED SYMBOLS*                LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   7
LIST EXAMPLE PRIOR TO MERGE
NAME          TYPE IN MODULE                    OM#  SLICE
-----
AUTO2         SD  AUTO2                1    1
AUTO21        SD  AUTO21               2    1
AUTO22        SD  AUTO22               3    1
    
```

--- END OF SECTION ---

```

BINDER V02.7A  *DUPLICATE SYMBOLS*            LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   8
LIST EXAMPLE PRIOR TO MERGE
NAME          TYPE IN MODULE                    OM#  SLICE TYPE IN MODULE
-----
    
```

--- END OF SECTION ---

*Lists after merging with output of merged modules*

```

BINDER V02.7A  *LOGICAL STRUCTURE*           LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   1
LIST EXAMPLE (MERGED-MODULES=YES)
SLICE TYPE  HSI   MMODE LEVEL STR# NAME                                T&D
-----
 1 LLM /7500 TU4K      0   1 COMPLEX2
 1 SUB /7500 TU4K      1   2  SUBLLM
 1 GM  /7500 TU4K      2   3  SUBLLM
1(OM ) /7500 TU4K      3   4  (AUTO2)
1(OM ) /7500 TU4K      3   5  (AUTO21)
 1 OM  /7500 TU4K      1   6  AUTO22
    
```

--- END OF SECTION ---

```

BINDER V02.7A  *SCOPE PATH INFORMATION*       LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   2
LIST EXAMPLE (MERGED-MODULES=YES)
PATH  STR#  PATHNAME
-----
    
```

--- END OF SECTION ---

```

BINDER V02.7A  *PROGRAM MAP*                 LLM:  COMPLEX2      DATE=2016-02-03 14:13:30 PAGE   3
LIST EXAMPLE (MERGED-MODULES=YES)
OBJ  NAME          ADDRESS  LENGTH  ATTRIB  SLICE  TYPE  RESOLVED  IN MODULE/FROM  ELEMENT#  OM#  SLICE  SATIS
-----
GM   SUBLLM                1
SD   SUBLLM  00000000 00000016 AA.....  1
(OM) (AUTO2)                1
(SD)(AUTO2)  00000000
(VC)(AUTO21)                00000010  AUTO21
VC   AUTO22                00000018  AUTO22
VC   AUTO23                FFFFFFFF
(OM) (AUTO21)                1
(SD)(AUTO21) 00000010
(VC)                AUTO2
OM   AUTO22                1
SD   AUTO22  00000018 00000006 AA.....  1
VC                AUTO2
    
```

--- END OF SECTION ---

BINDER V02.7A \*COMMON LIST\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 4

LIST EXAMPLE (MERGED-MODULES=YES)  
 NAME ADDRESS LENGTH ATTRIB TYPE IN MODULE OM# SLICE SATIS  
 --- END OF SECTION ---

BINDER V02.7A \*UNRESOLVED REFERENCES\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 5

LIST EXAMPLE (MERGED-MODULES=YES)  
 VC AUTO23  
 --- END OF SECTION ---

BINDER V02.7A \*UNRESOLVED LONG NAMES\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 6

LIST EXAMPLE (MERGED-MODULES=YES)  
 INDEX NAME  
 --- END OF SECTION ---

BINDER V02.7A \*SORTED SYMBOLS\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 7

LIST EXAMPLE (MERGED-MODULES=YES)  
 NAME TYPE IN MODULE OM# SLICE  
 (AUTO2) (SD)(AUTO2) 2 1  
 (AUTO21) (SD)(AUTO21) 3 1  
 AUTO22 SD AUTO22 4 1  
 SUBLLM SD SUBLLM 1 1  
 --- END OF SECTION ---

BINDER V02.7A \*DUPLICATE SYMBOLS\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 8

LIST EXAMPLE (MERGED-MODULES=YES)  
 NAME TYPE IN MODULE OM# SLICE TYPE IN MODULE OM# SLICE  
 --- END OF SECTION ---

*Lists after merging without merged modules*

BINDER V02.7A \*LOGICAL STRUCTURE\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 1

LIST EXAMPLE (MERGED-MODULES=NO)  
 SLICE TYPE HSI MMODE LEVEL STR# NAME T&D  
 --- END OF SECTION ---

BINDER V02.7A \*SCOPE PATH INFORMATION\* LLM: COMPLEX2 DATE=2016-02-03 14:13:30 PAGE 2

LIST EXAMPLE (MERGED-MODULES=NO)  
 PATH STR# PATHNAME  
 --- END OF SECTION ---

```

BINDER V02.7A   *PROGRAM MAP*                LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   3
LIST EXAMPLE (MERGED-MODULES=NO)
OBJ  NAME      ADDRESS    LENGTH    ATTRIB  SLICE  TYPE  RESOLVED  IN  MODULE/FROM  ELEMENT#  OM#  SLICE  SATIS
----  ---      -
GM   SUBLLM                    1
SD   SUBLLM    00000000  00000016  AA.....  1
VC   AUTO22                    00000018  AUTO22    2    1 SLICE
VC   AUTO23                    FFFFFFFF          UNRES
OM   AUTO22                    1
SD   AUTO22    00000018  00000006  AA.....  1
                                VC          SUBLLM          1    1 SLICE
                                --- END OF SECTION ---
    
```

```

BINDER V02.7A   *COMMON LIST*                LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   4
LIST EXAMPLE (MERGED-MODULES=NO)
NAME      ADDRESS    LENGTH    ATTRIB  TYPE  IN  MODULE      OM#  SLICE  SATIS
-----  -
                                --- END OF SECTION ---
    
```

```

BINDER V02.7A   *UNRESOLVED REFERENCES*      LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   5
LIST EXAMPLE (MERGED-MODULES=NO)
VC AUTO23
                                END OF SECTION ---
    
```

```

BINDER V02.7A   *UNRESOLVED LONG NAMES*      LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   6
LIST EXAMPLE (MERGED-MODULES=NO)
INDEX NAME
-----
                                --- END OF SECTION ---
    
```

```

BINDER V02.7A   *SORTED SYMBOLS*            LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   7
LIST EXAMPLE (MERGED-MODULES=NO)
NAME      TYPE  IN  MODULE      OM#  SLICE
-----  -
AUTO22    SD   AUTO22          2    1
SUBLLM    SD   SUBLLM          1    1
                                --- END OF SECTION ---
    
```

```

BINDER V02.7A   *DUPLICATE SYMBOLS*          LLM:  COMPLEX2    DATE=2016-02-03 14:13:30 PAGE   8
LIST EXAMPLE (MERGED-MODULES=NO)
NAME      TYPE  IN  MODULE      OM#  SLICE  TYPE  IN  MODULE      OM#  SLICE
-----  -
NAME      TYPE  IN  MODULE      OM#  SLICE  TYPE  IN  MODULE      OM#  SLICE
-----  -
                                --- END OF SECTION ---
    
```

**List example: Modules are included from SYSPLAMALT**

The example below shows how the INPUT INFORMATION section appears when modules are included from a SYSPLAMALT variable.

The actual library from which the modules are included can be determined thanks to the FILE ID columns in both parts of the INPUT INFORMATION section.

```
/sysplamalt-bnd1=
'(bnd.bnd.pl.in.llmplamalt1,bnd.bnd.pl.in.llmplamalt2,bnd.bnd.pl.in.llmplamalt3)'
...
/start-binder
//start-llm-creation internal-name=llmplamalt
//include-modules library=sysplamalt-bnd1,element=module11
//include-modules element=module21
//include-modules element=module22
//include-modules library=bnd.pl.in,element=om5
//save-llm library=llmplamalt.pl,map=yes
//end
```

*First part of the INPUT INFORMATION section*

BINDER	V02.7A	*INPUT INFORMATION*	LLM:	LLMPLAMALT	DATE=2016-02-04 11:41:39	PAGE	9	
	NAME		TYPE	VERSION	DATE	FILE ID	BIND	ELEM#
ELEMENT:	MODULE11		R	001	1989-07-25	1.1	EXPL	1
MODULES								
1	MODULE11		R	001	1989-07-25	1.2	EXPL	2
ELEMENT:	MODULE21							
MODULES								
2	MODULE21		R	001	1989-07-25	1.2	EXPL	3
ELEMENT:	MODULE22							
MODULES								
3	MODULE22		R	@	2008-09-30	2	EXPL	4
ELEMENT:	OM5							
MODULES								
4	OM5							

--- END OF SECTION ---

**FILE ID**

Identifier for the library from which the element is included.

If the identifier has the format 'x.y', it refers to one of the libraries specified in a SYSPLAMALT variable.

It provides the link with the second part of the section.

*Second part of the INPUT INFORMATION section*

```

BINDER V02.7A      *LINKNAME  CONVERSION*      LLM:  LLMPLAMALT  DATE=2016-02-04 11:41:39 PAGE  10
FILE ID           LINKNAME      FILE TYPE      FILENAME
-----
      1                PLAM          LLMPLAMALT-BND1
      1.1              :IOSN:$BINDER.BND.BND.PL.IN.LLMPLAMALT1
      1.2              :IOSN:$BINDER.BND.BND.PL.IN.LLMPLAMALT2
      2                PLAM          :IOSN:$BINDER.BND.PL.IN

```

```

--- END OF SECTION ---

```

**FILE ID**

Identifier for the input source.

It provides the link with the first part of the section.

If the identifier has the format 'x.y', it refers to one of the libraries specified in a SYSPLAMALT variable.

The corresponding SYSPLAMALT variable can be identified some lines before with the FILE ID 'x'.

---

## 5 BINDER run

A **BINDER run** is a sequence of statements beginning after the BINDER load call and ending with the END statement.

### 5.1 Calling and terminating BINDER

BINDER is loaded and started by means of the following command:

<b>START-BINDER</b>
<b>VERSION = *STD</b> , <b>MONJV = *NONE</b> / <filename 1..54 without-gen-vers> , <b>CPU-LIMIT = *JOB-REST</b> / <integer 1..32767 seconds>

#### **VERSION =**

Product version of the BINDER to be started.

Only the value \*STD is currently supported.

#### **VERSION = \*STD**

No product version is explicitly specified. The product version is selected as follows:

1. The version predefined with the /SELECT-PRODUCT-VERSION command.
2. The highest BINDER version installed with IMON.

#### **MONJV =**

Specifies a monitoring job variable for monitoring the BINDER run.

#### **MONJV = \*NONE**

No monitoring job variable is used.

#### **MONJV = <full-filename 1..54 without-gen-vers>**

Name of the job variable to be used.

**CPU-LIMIT =**

Maximum CPU time, in seconds, that the program may use.

**CPU-LIMIT = JOB-REST**

The remaining CPU time should be used for the job.

**CPU-LIMIT = <integer 1..32767 seconds>**

Only the specified time should be used.

After the program load message, BINDER expects the input of statements from the system file SYSDTA (see [page 138ff](#)).

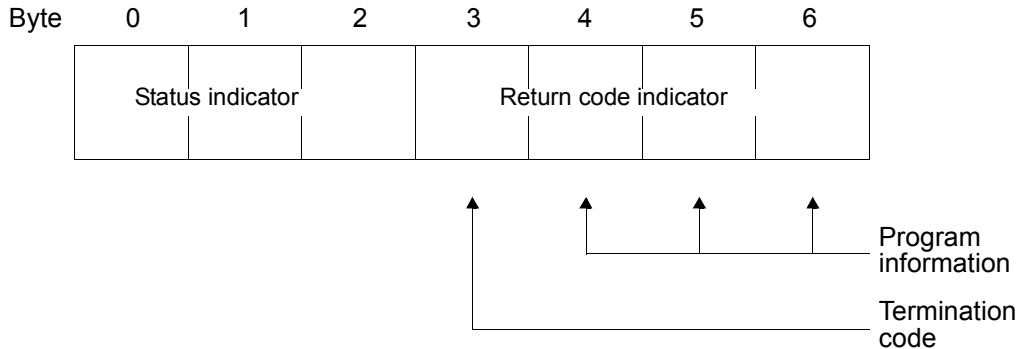
The END statement terminates the BINDER run.



## 5.2 Monitoring the BINDER run with job variables

The user can use a program-monitoring job variable (JV) for the execution of BINDER (see the “Job Variables” manual [6]). This presupposes that the software product JV is installed.

The program-monitoring job variable is 7 bytes long and has the following structure:



The first 3 bytes of the job variable (bytes 0-2) contain the **status indicator**. This reflects the current status of the BINDER run and may have the following values:

- \$R BINDER running (START-BINDER)
- \$T BINDER run terminated successfully
- \$A BINDER run aborted due to error

The following 4 bytes (bytes 3-6) contain the **return code indicator**. The return code indicator consists of the **termination code** (byte 3) and the **program information** (bytes 4-6).

### *Termination code*

- 0 BINDER run terminated without errors.
- 1 BINDER run terminated without errors. Warning messages were output.
- 2 BINDER run terminated with errors. The LLM contains errors or is not completely linked.
- 3 BINDER run terminated due to a serious error.

*Program information*

- 000 BINDER run terminated without errors. The LLM was linked without errors.
- 001 BINDER run terminated without errors. Warning messages were output.
- 002 BINDER run terminated without errors. Some unresolved external references were not deresolved.
- 003 BINDER run terminated with errors. Errors detected during syntax checking of the statements.
- 004 BINDER run terminated with errors. Certain recoverable errors were detected.
- 005 BINDER run terminated with errors. Errors in the input or user errors were detected. No LLM was linked, or the linked LLM may not be used.
- 006 BINDER run terminated with errors (internal BINDER errors). No LLM was linked, or the linked LLM may not be used.

The following table shows the relationship between the different values for the job variable and the severity classes. The value of the job variable corresponds to the *highest* severity class that occurred during the BINDER run.

\$T	0	000	INFORMATION
\$T	1	001	WARNING
\$T	1	002	UNRESOLVED EXTERNS
\$A	2	003	SYNTAX ERROR
\$A	2	004	RECOVERABLE ERROR
\$A	3	005	FATAL ERROR
\$A	3	006	INTERNAL ERROR

---

## 6 Subroutine interface

### 6.1 BINDER macro

BINDER is called as a subroutine from a main program by means of the BINDER macro; the statements to BINDER can be input in the following ways:

- Input from SYSDTA  
After BINDER is called, it requests the statements in succession from SYSDTA. It returns control to the main program when the END statement is entered.
- Input from the main program  
Each time BINDER is called, the main program issues *one* statement. One call is required for each statement.

#### Format and operand description

Operation	Operands
BINDER	[ BNDSTMT = <u>NULL</u> / addr ] [ ,MF = <u>S</u> / C / D / E / L / M ] [ ,PARAM = opaddr / (r) ] [ ,PREFIX = <u>P</u> / x ] [ ,MACID = <u>BND</u> / yyy ]

**BNDSTMT** Specifies whether the statements are input to BINDER from SYSDTA or from the main program.

**=NULL** The statements are input from SYSDTA.

**=addr** The statements are input from the main program. “addr” denotes the symbolic address (name) of a parameter area used for inputting one statement (see next page).

---

MF	Controls the form of macro expansion (see the “Executive Macros” manual [7]).
= <u>S</u>	S form
=C	C form (CSECT only)
=D	D form (DSECT only)
=E	E form (instruction code only) Generates the necessary instructions for execution of the macros.
=L	L form (data only)
=M	M form
PARAM	For E form and M form only.
=opaddr	Operand list, previously created with the control operand MF=L, whose address should be specified with “opaddr”.
=(r)	Operand list, previously created with the control operand MF=L, whose address should be specified in register (r).
PREFIX=x	Controls the generation of names (for C form, D form and M form only). <i>One</i> letter should be specified for “x”. This will be used as the first letter of all symbolic names. The remaining characters of the name are not changed. Default value is “P”.
MACID=yyy	Controls the generation of names (for C form and M form only). <i>Three</i> letters should be specified for “yyy”. These determine the second through fourth characters of the symbolic names. Default value is “BND”.

*Notes*

- The *macro* BINDER is supplied in the macro library \$.SYSLIB.BINDER.027.
- The *module* BINDER is supplied in the program library \$.SYSLNK.BINDER.027. Because the BINDER module is maintained in this library, it is more advantageous to load this module dynamically in the main program with the BIND macro since the calling program can then always use the current BINDER module (see following examples). The S form of the BINDER macro should not be used since it generates a V-type constant V(BINDER).
- If the statements are input by the calling program, the same parameter list must always be used. BINDER stores information that is used for the next call in the parameter list. The parameter list should therefore be set up with MF=L when BINDER is first called, and modified with MF=M on subsequent calls (see “[Example 2](#)” on page 180).
- BINDER utilizes fields in the parameter list for internal information. Therefore the values in the parameter list should not be modified directly, but implicitly by the BINDER macro with MF=M.

**Structure of the parameter area**

The parameter area for the input of a statement has the following structure:

0-1	2	Length of statement, including 4-byte record length field (n+4)
2-3	2	Reserved (ignored)
4-n	any	Statement with length n. The statement is input without syntax checking in the specified format to SDF. The user must therefore decide whether or not lowercase letters are permissible.

**Register conventions**

When calling BINDER as a subroutine, the following register conventions must be observed:

Register 1: Address of the parameter list of the BINDER macro

Register 14: Return address to the main program

Register 15: Address of the entry point to BINDER

### Return information and error flags

BINDER returns a **return code (RC)** for each macro call, containing information about execution of the macro. The return code is transferred in the *least significant* byte of the field MAINCODE in the standard header (see the “Executive Macros” manual [7]). The values denote hexadecimal constants. The return code issued corresponds to the highest severity class that occurred on calling BINDER.

Furthermore, subcode1 shows whether the BINDER macro was terminated normally or abnormally.

Subcode2 contains the maximum permissible error weight in case the abnormal termination was caused by this error weight being exceeded. Otherwise subcode2 contains the value X'FF'.

The following tables show the return code values, the subcode1 values and the meaning of these values. They also give the associated symbolic names. The user can specify the first character in the symbolic names with the PREFIX=x operand, and the second through fourth characters with the MACID=yyy operand. The default values are PREFIX=P and MACID=BND.

X' 00'	OK	No error
X' 01'	INFO	Error of severity class INFORMATION
X' 02'	WARN	Error of severity class WARNING
X' 03'	UNRE	Error of severity class UNRESOLVED EXTERNS
X' 04'	SYNT	Error of severity class SYNTAX ERROR
X' 05'	RECO	Error of severity class RECOVERABLE ERROR
X' 06'	FATA	Error of severity class FATAL ERROR
X' 07'	INTE	Error of severity class INTERNAL ERROR

Subcode1	Symbolic name xyyy...	Meaning
X'FF'	NORM	Normal termination
X'40'	ABN	Abnormal termination

## 6.2 Examples

### Example 1

A routine BNDCALL1 loads BINDER dynamically with the BIND macro from the program library \$.SYSLNK.BINDER.027. BINDER is called with the BINDER macro and the BINDER statements are requested from SYSDTA. The routine BNDCALL1 is called as a subroutine by the main program PROG1.

#### Source listing of BNDCALL1

```

BNDCALL1 CSECT
        USING BNDCALL1,15
        STM 0,15,SAVEREG ----- (1)
        DROP 15
        LR 10,15
        USING BNDCALL1,10
        L 1,BINDER@ ----- (2)
        LTR 1,1
        BNZ NOT1CALL ----- (3)
        BIND MF=E,PARAM=BINDPL ----- (4)
        LA 2,BINDPL
        USING BINDDS,2 ----- (5)
        CLI XBINSR2,XBINPART ----- (6)
        BNL BINDERR ----- (7)
NOT1CALL DS OH
        USING BNDDS,1 ----- (8)
        LA 1,BNDPL ----- (9)
        L 15,BINDER@ ----- (10)
        BALR 14,15 ----- (11)
        CLI YBNDMRET+1,YBNDSYNT ----- (12)
        BNL BNDERROR ----- (13)
RETURN  DS OH
        LM 0,15,SAVEREG ----- (14)
        BR 14 ----- (15)
BINDERR DS OH
        B RETURN
BNDERROR DS OH
        B RETURN
SAVEREG DS 16F
BINDER@ DC A(0)
BNDPL  BINDER MF=L ----- (16)
MODNAM DC CL32'BINDER '
MODLIB DC CL54'$.SYSLNK.BINDER.027 '
BINDPL BIND MF=L,SYMBOL@=MODNAM,LIBNAM@=MODLIB,SYMBLAD=BINDER@ ----- (17)
        LTORG
BNDDS  BINDER MF=D,PREFIX=Y ----- (18)

```

```
BINDDS  BIND  MF=D,PREFIX=X _____ (19)
        END
```

*Source listing of PROG1*

```
PROG1   START
        BALR  3,0
        USING *,3
        L     15,=V(BNDCALL1) _____ (20)
        BALR  14,15
        WROUT AUS,FEHLER _____ (21)
FEHLER  TERM
AUS     DC    Y(AUSE-AUS)
        DS    CL3
        DC    C'BINDER CALL TERMINATED'
AUSE    EQU   *
        END
```

- (1) All registers are saved in the save area of BNDCALL1.
- (2) The field BINDER@ is checked. The BIND macro uses it to pass the start address of the dynamically loaded module BINDER (see (4)).
- (3) If an address was passed in the field BINDER@, the module BINDER is already loaded and the BIND call is skipped.
- (4) The BIND macro is called in its E form. Only the instruction code is therefore generated at this point in the program. The associated parameter list is set up at the symbolic address BINDPL (see (17)) by a BIND call with MF=L. The operand values specified in it cause the BIND macro to perform the following actions on program execution:
  - dynamically load the module BINDER (SYMBOL@=MODNAM) from the library \$.SYSLNK.BINDER.027 (LIBNAM@=MODLIB)
  - pass the start address of the module BINDER in the field BINDER@ (SYMBLAD=BINDER@)
  - after loading of the module BINDER in the calling routine BNDCALL1, continue with the instruction following the BIND macro (default value BRANCH=NO).
- (5) Register 2 is assigned to the assembler as the base address register for addressing the DSECT for the parameter list of the BIND macro, that is generated at the symbolic address BINDDS by a BIND call with MF=D.



- (6) After execution of the BIND macro, the field XBINSR2 of the standard header containing the SUBCODE2 is compared with the SUBCODE2 XBINPART of the standard header that establishes error-free execution of the BIND macro. The names XBINSR2 and XBINPART originate from the DSECT that was generated under the symbolic address BINDDS by a BIND call with MF=D and PREFIX=X (see (17)). This DSECT describes the structure of the parameter list of the BIND macro. The symbolic names of the DSECT can be used for addressing within the parameter list after the assigned base address register (register 2 in this case) has been loaded with the start address of the parameter list (BINDPL here).
- (7) Branch to the error exit BINDERR if the BIND macro is errored in its execution or is not executed.
- (8) Register 1 is assigned to the assembler as the base address register for addressing the DSECT for the parameter list of the BINDER macro, that is generated at the symbolic address BNDDS (see (18)) by a BINDER call with MF=D.
- (9) Register 1 is loaded with the address of the parameter list of the BINDER macro. The parameter list is generated at the symbolic address BNDPL (see (16)) by a BINDER call with MF=L. The default value BNDSTMT=NULL specifies that the BINDER statements are requested by BINDER from SYSDTA.
- (10) Register 15 is loaded with the start address of the module BINDER, that was passed by the BIND macro in the field BINDER@.
- (11) Call for the module BINDER that requests the BINDER statements from SYSDTA. After input of the END statement, the BINDER run is terminated and control returns to the routine BNDCALL1.
- (12) After execution of the BINDER macro, the field YBNMRET+1 of the standard header that denotes the least significant byte of the MAINCODE is compared with the return code YBNDSYNT that specifies the severity class SYNTAX ERROR. The names YBNMRET and YBNDSYNT originate from the DSECT that was generated under the symbolic address BNDDS by a BINDER call with MF=D and PREFIX=Y (see (18)). This DSECT describes the structure of the parameter list of the BINDER macro. The symbolic names of the DSECT can be used for addressing within the parameter list after the assigned base address register (register 1 in this case) has been loaded with the start address of the parameter list (BNDPL here).
- (13) Branch to the error exit BINDERERR if errors of severity class SYNTAX ERROR or higher have occurred during execution of the BINDER macro.
- (14) Restore original contents to all saved registers.
- (15) Return to the user program that called the routine BNDCALL1.
- (16) The call of the BINDER macro in its L form creates the parameter list for the call of the BINDER module (see (9)) and supplies it with values.

- (17) The L form of the BIND macro creates the parameter list for the BIND macro (see (4)).
- (18) The BINDER macro with MF=D generates a DSECT that describes the structure of the parameter list of the BINDER macro. The PREFIX=Y operand causes all symbolic names in this DSECT (field names and equates) to begin with the letter Y.
- (19) The BIND macro with MF=D generates a DSECT that describes the structure of the parameter list of the BIND macro. The PREFIX=X operand causes all symbolic names in this DSECT (field names and equates) to begin with the letter X.
- (20) The routine BNDCALL1 is called by the user program PROG1 as a subroutine.
- (21) A message to SYSOUT indicates that the routine BNDCALL1 has been terminated.

### Runtime listing

```

/set-file-link link-name=altlib,file-name=$.syslib.binder.027 _____ (1)
/start-assembly
% BLS0500 PROGRAM 'ASSEMBH', VERSION '1.2C00' OF '2002-03-06' LOADED
% BLS0552 COPYRIGHT (C) 2016 FUJITSU TECHNOLOGY SOLUTIONS GMBH.
  ALL RIGHTS RESERVED
% ASS6010 V01.2C00 OF BS2000 ASSEMBH  READY
///compile source=src.bndcall1,macro-library=*link(link-name=altlib),-
///      module-library=bndlib _____ (2)
% ASS6011 ASSEMBLY TIME: 1458 MSEC
% ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
% ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
% ASS6006 LISTING GENERATOR TIME: 867 MSEC
///compile source=src.prog1,module-library=*omf _____ (3)
% ASS6011 ASSEMBLY TIME: 697 MSEC
% ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
% ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
% ASS6006 LISTING GENERATOR TIME: 260 MSEC
///end
% ASS6012 END OF ASSEMBH
/set-file-link link-name=blslib01,file-name=bndlib
/start-program from-file=*module(library=*omf,element=prog1, -
/      run-mode=advanced(alternate-libraries=yes)) _____ (4)
% BLS0517 MODULE 'PROG1' LOADED
///start-llm-creation internal-name=llm1 _____ (5)
///include-modules library=bndlib,element=(mod1,mod2)
///save-llm library=bndlib
% BND1501 LLM FORMAT : '1'
///end
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'OK'
BINDER CALL TERMINATED _____ (6)

```

- (1) The macro library \$.SYSLIB.BINDER.027 that contains the BINDER macro is assigned to the assembler as ALTLIB.
- (2) The routine BNDCALL1 is compiled and the object module BNDCALL1 is stored in program library BNDLIB.
- (3) The user program PROG1 is compiled.
- (4) The program library BNDLIB containing the module BNDCALL1 is assigned as the alternative library.
- (5) DBL is called in order to load and start the module PROG1. The module PROG1 is fetched from the EAM object module library. External references are resolved from the alternative library.
- (6) The module BINDER is called by BNDCALL1 and requests the BINDER statements from SYSDTA. The END statement terminates the BINDER run.
- (7) The program PROG1 issues a message indicating that the routine BNDCALL1 has been terminated.

**Example 2**

A routine BNDCALL2 loads BINDER dynamically with the BIND macro from the program library \$.SYSLNK.BINDER.027. BINDER is called with the BINDER macro. The first two BINDER statements are entered by the routine BNDCALL2. BINDER then requests further BINDER statements from SYSDTA. The routine BNDCALL2 is called as a subroutine by a main program PROG2.

*Source listing of BNDCALL2*

```

BNDCALL2 CSECT
        USING  BNDCALL2,15
        STM   0,15,SAVEREG  _____ (1)
        DROP  15
        LR    10,15
        USING BNDCALL2,10
        L     1,BINDER@ _____ (2)
        LTR   1,1
        BNZ   NOT2CALL _____ (3)
        BIND  MF=E,PARAM=BINDPL _____ (4)
        LA   2,BINDPL
        USING BINDDS,2 _____ (5)
        CLI   XBINSR2,XBINPART _____ (6)
        BNL   BINDERR _____ (7)
NOT2CALL DS   0H
        LA   1,BNDPL _____ (8)
        USING BNDDS,1 _____ (9)
        L    15,BINDER@ _____ (10)
        BALR 14,15 _____ (11)
        CLI  YBNDMRET+1,YBNDSYNT _____ (12)
        BL   CAL2STMT _____ (13)
        CLI  YBNDMRET+1,YBNDFATA _____ (14)
        BNL  RETURN _____ (15)
CAL2STMT BINDER MF=M,BNDSTMT=STMT2LG,PREFIX=Y _____ (16)
        BALR 14,15 _____ (17)
        BINDER MF=M,BNDSTMT=NULL,PREFIX=Y _____ (18)
        BALR 14,15 _____ (19)
        CLI  YBNDMRET+1,YBNDSYNT _____ (20)
        BNL  BNDERROR _____ (21)
RETURN   DS   0H
        LM   0,15,SAVEREG _____ (22)
        BR   14 _____ (23)
BINDERR  DS   0H
        B    RETURN
BNDERROR DS   0H
        B    RETURN
SAVEREG  DS   16F
BINDER@  DC   A(0)

```

```

STMT1LG DC H'40' _____ (24)
STMT1UN DC H'0'
STMT1 DC CL36'START-LLM-CREATION INTERNAL-NAME=LLM'
STMT2LG DC H'56' _____ (25)
STMT2UN DC H'0'
STMT2 DC CL52'INCLUDE-MODULES LIBRARY=BNDLIB,ELEMENT=(MOD1,MOD2) '
BNDPL BINDER MF=L,BNDSTMT=STMT1LG _____ (26)
MODNAM DC CL32'BINDER '
MODLIB DC CL54'$.SYSLNK.BINDER.027 '
BINDPL BIND MF=L,SYMBOL@=MODNAM,LIBNAM@=MODLIB,SYMBLAD=BINDER@ _____ (27)
        LTORG
BNDDS BINDER MF=D,PREFIX=Y _____ (28)
BINDDS BIND MF=D,PREFIX=X _____ (29)
        END

```

*Source listing of PROG2*

```

PROG2 START
        BALR 3,0
        USING *,3
        L 15,=V(BNDCALL2) _____ (30)
        BALR 14,15
        WROUT AUS,FEHLER _____ (31)
FEHLER TERM
AUS DC Y(AUSE-AUS)
        DS CL3
        DC C'BINDER CALL TERMINATED'
AUSE EQU *
        END

```

- (1) All registers are saved in the save area of BNDCALL2.
- (2) The field BINDER@ is checked. The BIND macro uses it to pass the start address of the dynamically loaded module BINDER (see 04).
- (3) If an address was passed in the field BINDER@, the module BINDER is already loaded and the BIND call is skipped.

- (4) The BIND macro is called in its E form. Only the instruction code is therefore generated at this point in the program. The associated parameter list is set up at the symbolic address BINDPL by a BIND call with MF=L. The operand values specified in it cause the BIND macro to perform the following actions on program execution:
  - dynamically load the module BINDER (SYMBOL@=MODNAM) from the library \$.SYSLNK.BINDER.027 (LIBNAM@=MODLIB)
  - pass the start address of the module BINDER in the field BINDER@ (SYMBLAD=BINDER@)
  - after loading of the module BINDER in the calling routine BNDCALL2, continue with the instruction following the BIND macro (default value BRANCH=NO).
- (5) Register 2 is assigned to the assembler as the base address register for addressing the DSECT for the parameter list of the BIND macro, that is generated at the symbolic address BINDDS by a BIND call with MF=D.
- (6) After execution of the BIND macro, the field XBINSR2 of the standard header containing the SUBCODE2 is compared with the SUBCODE2 XBINPART of the standard header that establishes error-free execution of the BIND macro. The names XBINSR2 and XBINPART originate from the DSECT that was generated under the symbolic address BINDDS by a BIND call with MF=D and PREFIX=X (see (27)). This DSECT describes the structure of the parameter list of the BIND macro. The symbolic names of the DSECT can be used for addressing within the parameter list after the assigned base address register (register 2 in this case) has been loaded with the start address of the parameter list (BINDPL here).
- (7) Branch to the error exit BINDERR if the BIND macro is errored in its execution or is not executed.
- (8) Register 1 is loaded with the address of the parameter list of the BINDER macro. The parameter list is generated at the symbolic address BNDPL by a BINDER call with MF=L. The value STMT1LG of the BNDSTMT operand specifies that the BINDER statement START-LLM-CREATION that is constructed in the parameter area starting at address STMT1LG is taken over.
- (9) Register 1 is assigned to the assembler as the base address register for addressing the DSECT for the parameter list of the BINDER macro, that is generated at the symbolic address BNDDS by a BINDER call with MF=D.
- (10) Register 15 is loaded with the start address of the module BINDER, that was passed by the BIND macro in the field BINDER@.
- (11) Call for the module BINDER that takes the first BINDER statement from the parameter area starting at address STMT1LG.

- (12) After execution of the BINDER macro, the field YBNDMRET+1 of the standard header that denotes the least significant byte of the MAINCODE is compared with the return code YBNDSYNT that specifies the severity class SYNTAX ERROR. The names YBNDMRET and YBNDSYNT originate from the DSECT that was generated under the symbolic address BNDDS by a BINDER call with MF=D and PREFIX=Y (see (26)). This DSECT describes the structure of the parameter list of the BINDER macro. The symbolic names of the DSECT can be used for addressing within the parameter list after the assigned base address register (register 1 in this case) has been loaded with the start address of the parameter list (BNDPL here).
- (13) Branch to the next BINDER call if no errors of severity class SYNTAX ERROR or higher have occurred during execution of the BINDER macro
- (14) The field YBNDMRET+1 of the standard header that denotes the least significant byte of the MAINCODE is compared with the return code YBNDFATA that specifies the severity class FATAL ERROR.
- (15) If errors of severity class FATAL ERROR or higher have occurred during execution of the BINDER macro, the routine BNDCALL2 branches to the user program PROG2 from which it was called.
- (16) The BINDER macro with MF=M modifies the parameter list that was set up by the previous BINDER macro with MF=L. The value STMT2LG of the BNDSTMT operand specifies that the BINDER statement INCLUDE-MODULES that is constructed in the parameter area starting at address STMT2LG is taken over.
- (17) Call for the module BINDER that takes the BINDER statement from the parameter area starting at address STMT2LG.
- (18) The BINDER macro with MF=M modifies the parameter list that was set up with MF=L. The value NULL of the BNDSTMT operand specifies that the following BINDER statements are requested from SYSDTA.
- (19) Call for the module BINDER that requests the BINDER statements from SYSDTA. On entry of the END statement, the BINDER run is terminated and control is returned to the routine BNDCALL2.
- (20) After execution of the BINDER macro, the field YBNDMRET+1 of the standard header that denotes the least significant byte of the MAINCODE is compared with the return code YBNDSYNT that specifies the severity class SYNTAX ERROR.
- (21) Branch to the error exit BINDERERR if errors of severity class SYNTAX ERROR or higher have occurred during execution of the BINDER macro.
- (22) Restore original contents to all saved registers.
- (23) Return to the user program that called the routine BNDCALL2.
- (24) Parameter area in which the START-LLM-CREATION statement is constructed.

- (25) Parameter area in which the INCLUDE-MODULES statement is constructed.
- (26) The BINDER macro with MF=L creates the parameter list for the call of the BINDER module (see (11)) and supplies it with values, e.g. the statement to be executed.
- (27) The BIND macro with MF=L creates the parameter list for the BIND macro (see(4)).
- (28) The BINDER macro with MF=D generates a DSECT that describes the structure of the parameter list of the BINDER macro. The PREFIX=Y operand causes all symbolic names in this DSECT (field names and equates) to begin with the letter Y.
- (29) The BIND macro with MF=D generates a DSECT that describes the structure of the parameter list of the BIND macro. The PREFIX=X operand causes all symbolic names in this DSECT (field names and equates) to begin with the letter X.
- (30) The routine BNDCALL2 is called by the user program PROG2 as a subroutine.
- (31) A message to SYSOUT indicates that the routine BNDCALL2 has been terminated.

#### *Runtime listing*

```

/set-file-link link-name=altlib,file-name=$.syslib.binder.027 _____ (1)
/start-program from-file=$assembh
% BLS0500 PROGRAM 'ASSEMBH', VERSION '1.2C00' OF '2002-03-06' LOADED
% BLS0552 COPYRIGHT (C) 2016 FUJITSU TECHNOLOGY SOLUTIONS GMBH.
  ALL RIGHTS RESERVED
% ASS6010 V01.2C00 OF BS2000 ASSEMBH  READY
///compile source=src.bndcall2,macro-library=*link(link-name=altlib),-
///module-library=bndlib _____ (2)
% ASS6011 ASSEMBLY TIME: 1486 MSEC
% ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
% ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
% ASS6006 LISTING GENERATOR TIME: 855 MSEC
///compile source=src.prog2,module-library=*omf _____ (3)
% ASS6011 ASSEMBLY TIME: 688 MSEC
% ASS6018 0 FLAGS, 0 PRIVILEGED FLAGS, 0 MNOTES
% ASS6019 HIGHEST ERROR-WEIGHT: NO ERRORS
% ASS6006 LISTING GENERATOR TIME: 255 MSEC
///end
% ASS6012 END OF ASSEMBH

```



```

/set-file-link link-name=bllib01,file-name=bndlib _____ (4)
/start-program from-file=*module(library=*omf,element=prog2,-
/ run-mode=advanced(alternate-libraries=yes)) _____ (5)
% BLS0517 MODULE 'PROG2' LOADED
%//include-modules element=mod3,type=(l,r) _____ (6)
%//show-map help-information=no,global-information=no, - _____ (7)
%// physical-structure=no,program-map=no,unresolved-list=no,-
%// input-information=no,output=*by-show-file
% BND1601 'FILE' MACRO PERFORMED ON 'BNDMAP.2016-02-17138.173535.2AB0'

```

```

15000000BINDER V02.7A *LOGICAL STRUCTURE* LLM: X (8)
15000001
15000002SLICE TYPE HSI LEVEL STR# NAME
15000003-----
15100000 1 LLM /7500 0 1 LLMX
15100001 1 GM /7500 1 2 MOD1
15100002 1 GM /7500 1 2 MOD2
15100003 1 GM /7500 1 2 MOD3
15900000
17000000BINDER V02.0A *SCOPE PATH INFORMATION* LLM: X
17000001
17000002PATH STR# PATHNAME
17000003-----
17900000

% SH00301 WARNING: END OF FILE REACHED
e I*SOF+ 1( 1)

```

```

%//save-llm library=bndlib _____ (9)
%//end _____ (10)
% BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'OK'
BINDER CALL TERMINATED _____ (11)

```

- (1) The macro library \$.SYSLIB.BINDER.027 that contains the BINDER macro is assigned to the assembler as ALTLIB.
- (2) The routine BNDCALL2 is compiled and the object module BNDCALL2 is stored in program library BNDLIB.
- (3) The user program PROG2 is compiled and the object module PROG2 is stored in the EAM object module file.
- (4) The program library BNDLIB containing the object module BNDCALL2 is assigned as the alternative library.

- (5) DBL is called in order to load and start the module PROG2. The module PROG2 is fetched from the EAM object module library. After the first two statements START-LLM-CREATION and INCLUDE-MODULES, that have been passed by BNDCALL2, have been processed, the module BINDER requests further BINDER statements from SYSDTA.
- (6) The INCLUDE-MODULES statement is read in from SYSDTA. This fetches the object module MOD3 from the current library BLSLIB01 and includes it in the LLM.
- (7) The SHOW-MAP statement is entered from SYSDTA. Only the logical structure list is to be output.
- (8) Logical structure list for the created LLM. The ISAM keys are shown in the first eight columns.
- (9) The SAVE-LLM statement stores the created LLM in the program library BNDLIB.
- (10) The END statement terminates the BINDER run.
- (11) The program PROG2 issues a message indicating that the routine BNDCALL2 has been terminated.

---

## 7 BINDER statements

All the statements for BINDER are described in this chapter.

### 7.1 Grouping of statements by function

The statements for BINDER can be grouped as follows depending on their function:

#### Creating, updating and saving an LLM

START-LLM-CREATION	Creates a current LLM in the BINDER work area.
START-LLM-UPDATE	Updates an LLM saved in a program library.
SAVE-LLM	Saves the current LLM from the BINDER work area to a program library.

#### Including, removing and replacing modules

INCLUDE-MODULES	Includes one or more modules in the current LLM.
REMOVE-MODULES	Removes one or more modules from the current LLM.
REPLACE-MODULES	Replaces one or more modules in the current LLM with new modules.

#### Merging modules

MERGE-MODULES	Merges all modules of a (sub-)LLM into one LLM which then contains only one prelinked module with a single CSECT.
---------------	-------------------------------------------------------------------------------------------------------------------

**Creating the logical structure of an LLM**

BEGIN-SUB-LLM-STATEMENTS	Defines the beginning of a sub-LLM within an LLM or a sub-LLM.
END-SUB-LLM-STATEMENTS	Defines the end of a sub-LLM within an LLM or a sub-LLM.

**Creating the physical structure of an LLM**

SET-USER-SLICE-POSITION	Defines the position of a slice in the physical structure of an LLM.
-------------------------	----------------------------------------------------------------------

**Modifying the attributes of LLMs and modules**

MODIFY-LLM-ATTRIBUTES	Modifies the attributes of an LLM.
MODIFY-MODULE-ATTRIBUTES	Modifies the attributes of the modules in the current LLM and can, for example, modify the logical structure of the LLM.

**Resolving external references by autolink**

RESOLVE-BY-AUTOLINK	Automatically includes modules in the current LLM that resolve unresolved external references.
SET-EXTERN-RESOLUTION	Declares how unresolved external references that cannot be resolved are to be handled.

**Handling symbols**

RENAME-SYMBOLS	Replaces the names of symbols in the current LLM with new names.
MODIFY-SYMBOL-ATTRIBUTES	Modifies the attributes of CSECTs and COMMONs in the current LLM.
MODIFY-SYMBOL-TYPE	Modifies the symbol types (EXTRN, WXTRN and VCON).
MODIFY-SYMBOL-VISIBILITY	Defines the extent to which program definitions (CSECTs) and entry points (ENTRYs) remain visible or are masked in the current LLM.

**Display functions**

SHOW-DEFAULTS	Displays the global defaults for a BINDER run.
SHOW-LIBRARY-ELEMENTS	Displays and checks library elements.
SHOW-SYMBOL-INFORMATION	Displays selected information about symbols.

**Controlling list output and error processing**

SHOW-MAP	Outputs lists containing information about the current LLM.
MODIFY-MAP-DEFAULTS	Modifies the default values for list output.
MODIFY-ERROR-PROCESSING	Specifies that the BINDER run will be terminated on the occurrence of errors of certain severity classes, and controls message output.
MODIFY-STD-DEFAULTS	Modifies the global defaults for a BINDER run.
START-STATEMENT-RECORDING	Records BINDER statements
STOP-STATEMENT-RECORDING	Terminates recording of BINDER statements

**Terminating the BINDER run**

END	Terminates the BINDER run.
-----	----------------------------

The SDF standard statements may also be specified. They (except for END) are not described in the present manual. A description of these statements can be found in the "Introductory Guide to the SDF Dialog Interface" [13].

## 7.2 Notes on the SDF user interface

The statements described in this manual are processed by the command processor SDF (System Dialog Facility). This supports various levels of guided and unguided dialog, enabling less experienced users to request help menus for the statements. Input errors can be corrected in a correction dialog. Detailed information on various options provided by SDF can be found in the "Introductory Guide to the SDF Dialog Interface" [13].

### Abbreviation of names

SDF permits inputs to be abbreviated in interactive and batch modes as well as in procedures, provided the abbreviations used are unambiguous within the related syntax environment. Note, however, that an abbreviation that is currently unambiguous could potentially become ambiguous at a later date, particularly if new functions are added to the product. For this reason, it is best to avoid abbreviations entirely, especially in procedures, or at the very least, to ensure that only guaranteed abbreviations be used. In the statement formats, these guaranteed abbreviations are shown in boldface.

Command and statement names, operands and keyword values may be abbreviated as follows:

- Complete name components may be omitted from right to left; the hyphen preceding the dropped name component is also omitted.
- Individual characters of a name component may be omitted from right to left.
- An asterisk (\*) preceding a keyword value is not considered a valid abbreviation for that value. As of SDF V4.0A, keyword values are always represented with a leading asterisk. The asterisk may be omitted only if there is no possible alternative variable operand value with a value range that includes the name of the keyword value. This form of abbreviation may be restricted due to extensions in later versions. For compatibility reasons, operand values that were previously represented without an asterisk are still accepted without the asterisk.

#### *Example of input*

Unabbreviated command format:

```
/MODIFY-SDF-OPTIONS SYNTAX-FILE=*NONE,GUIDANCE=*MINIMUM
```

Abbreviated command format:

```
/MOD-SDF-OPT SYN-F=*NO,GUID=*MIN
```

The guaranteed abbreviations are only intended as recommendations for abbreviated input; they may not always be the shortest possible input in your syntax environment. They are, however, clear and easy to understand and are designed to remain unique in the long term.

In some cases, an additional abbreviation is documented in the manual next to the command or statement name. This abbreviation is implemented as an alias for the command or statement name and is guaranteed in the long term. The alias consists of a maximum of 8 letters (A...Z) that are derived from the command or statement name. Aliases cannot be abbreviated further.

### Default values

Most operands are optional, i.e. need not be explicitly specified. Such operands are preset to a specific operand value, the so-called default value. The default value for each operand is shown in the syntax underscored. If an optional operand is not explicitly specified, its default value is automatically inserted when executing the command or statement.

### Positional operands

SDF permits operands to be specified either as keyword operands or as positional operands. However, it is quite possible that the positions of operands may change in future versions of the product. It is therefore advisable to avoid the use of positional operands, especially in procedures.

## 7.2.1 SDF syntax description

This syntax description is valid for SDF V4.5A. The syntax of the SDF command/statement language is explained in the following three tables.

*Table 1: Notational conventions*

The meanings of the special characters and the notation used to describe command and statement formats are explained in [table 1](#).

*Table 2: Data types*

Variable operand values are represented in SDF by data types. Each data type represents a specific set of values. The number of data types is limited to those described in [table 2](#).

The description of the data types is valid for the entire set of commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.

*Table 4: Suffixes for data types*

Data type suffixes define additional rules for data type input. They contain a length or interval specification and can be used to limit the set of values (suffix begins with *without*), extend it (suffix begins with *with*), or declare a particular task mandatory (suffix begins with *mandatory*). The following short forms are used in this manual for data type suffixes:

cat-id	cat
completion	compl
correction-state	corr
generation	gen
lower-case	low
manual-release	man
odd-possible	odd
path-completion	path-compl
separators	sep
temporary-file	temp-file
underscore	under
user-id	user
version	vers
wildcard-constr	wild-constr
wildcards	wild

The description of the 'integer' data type in [table 4](#) contains a number of items in italics; the italics are not part of the syntax and are only used to make the table easier to read.

For special data types that are checked by the implementation, [table 4](#) contains suffixes printed in italics (see the *special* suffix) which are not part of the syntax.

The description of the data type suffixes is valid for the entire set of commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.



## Metasyntax

Representation	Meaning	Examples
UPPERCASE LETTERS	Uppercase letters denote keywords (command, statement or operand names, keyword values) and constant operand values. Keyword values begin with *.	<b>HELP-SDF</b>
<b>UPPERCASE LETTERS</b> in boldface	Uppercase letters printed in boldface denote guaranteed or suggested abbreviations of keywords.	<b>SCREEN-STEPS = *NO</b>
=	The equals sign connects an operand name with the associated operand values.	<b>GUIDANCE-MODE = *YES</b>
< >	Angle brackets denote variables whose range of values is described by data types and suffixes (see Tables 2 and 4).	<b>GUIDANCE-MODE = *NO</b>
<u>Underscoring</u>	Underscoring denotes the default value of an operand.	<b>SYNTAX-FILE = &lt;filename 1..54&gt;</b>
/	A slash serves to separate alternative operand values.	<b>GUIDANCE-MODE = *NO</b>
(...)	Parentheses denote operand values that initiate a structure.	<b>NEXT-FIELD = *NO / *YES</b>
[ ]	Square brackets denote operand values which introduce a structure and are optional. The subsequent structure can be specified without the initiating operand value.	<b>,UNGUIDED-DIALOG = *YES (...) / *NO</b>
Indentation	Indentation indicates that the operand is dependent on a higher-ranking operand.	<b>SELECT = [*BY-ATTRIBUTES](...)</b>
		<b>GUIDED-DIALOG = *YES (...)</b> <b>*YES(...)</b>   <b>SCREEN-STEPS = *NO / *YES</b>

Table 1: Metasyntax (part 1 of 2)

Representation	Meaning	Examples
<p>,</p> <p>list-poss(n):</p> <p>Alias:</p>	<p>A vertical bar identifies related operands within a structure. Its length marks the beginning and end of a structure. A structure may contain further structures. The number of vertical bars preceding an operand corresponds to the depth of the structure.</p> <p>A comma precedes further operands at the same structure level.</p> <p>The entry "list-poss" signifies that a list of operand values can be given at this point. If (n) is present, it means that the list must not have more than n elements. A list of more than one element must be enclosed in parentheses.</p> <p>The name that follows represents a guaranteed alias (abbreviation) for the command or statement name.</p>	<pre> SUPPORT = *TAPE(...)     *TAPE(...)                     VOLUME = *ANY(...)             *ANY(...)                             ...     , GUIDANCE-MODE = *NO / *YES ,SDF-COMMANDS = *NO / *YES  list-poss: *SAM / *ISAM  list-poss(40): &lt;structured-name 1..30&gt;  list-poss(256): *OMF / *SYSLST(...) /                 &lt;filename 1..54&gt;  HELP-SDF      Alias: HPSDF         </pre>

Table 1: Metasyntax (part 2 of 2)

**Data types**

<b>Data type</b>	<b>Character set</b>	<b>Special rules</b>
alphanum-name	A...Z 0...9 \$, #, @	
cat-id	A...Z 0...9	Not more than 4 characters; must not begin with the string PUB
command-rest	freely selectable	
composed-name	A...Z 0...9 \$, #, @ hyphen period catalog ID	Alphanumeric string that can be split into multiple substrings by means of a period or hyphen. If a file name can also be specified, the string may begin with a catalog ID in the form :cat: (see data type filename).
c-string	EBCDIC character	Must be enclosed within single quotes; the letter C may be prefixed; any single quotes occurring within the string must be entered twice.
date	0...9 Structure identifier: hyphen	Input format: yyyy-mm-dd  jjjj: year; optionally 2 or 4 digits mm: month tt: day
device	A...Z 0...9 hyphen	Character string, max. 8 characters in length, corresponding to a device available in the system. In guided dialog, SDF displays the valid operand values. For notes on possible devices, see the relevant operand description.
fixed	+, - 0...9 period	Input format: [sign][digits].[digits]  [sign]: + or - [digits]: 0...9  must contain at least one digit, but may contain up to 10 characters (0...9, period) apart from the sign.

Table 2: Data types (part 1 of 5)

Data type	Character set	Special rules
filename	A...Z 0...9 \$, #, @ hyphen period	<p>Input format:</p> $[:cat:][\$user.] \left\{ \begin{array}{l} \text{file} \\ \text{file(no)} \\ \text{group} \end{array} \right\}$ $\text{group} \left\{ \begin{array}{l} (*abs) \\ (+rel) \\ (-rel) \end{array} \right\}$ <p>:cat: optional entry of the catalog identifier; character set limited to A...Z and 0...9; maximum of 4 characters; must be enclosed in colons; default value is the catalog identifier assigned to the user ID, as specified in the user catalog.</p> <p>\$user. optional entry of the user ID; character set is A...Z, 0...9, \$, #, @; maximum of 8 characters; first character cannot be a digit; \$ and period are mandatory; default value is the user's own ID.</p> <p>\$. (special case) system default ID</p> <p>file file or job variable name; may be split into a number of partial names using a period as a delimiter: name<sub>1</sub>[.name<sub>2</sub>[...]] name<sub>i</sub> does not contain a period and must not begin or end with a hyphen; file can have a maximum length of 41 characters; it must not begin with a \$ and must include at least one character from the range A...Z.</p>

Table 2: Data types (part 2 of 5)

Data type	Character set	Special rules
filename (contd.)		<p>#file (special case) @file (special case) # or @ used as the first character indicates temporary files or job variables, depending on system generation.</p> <p>file(no) tape file name no: version number; character set is A...Z, 0...9, \$, #, @. Parentheses must be specified.</p> <p>group name of a file generation group (character set: as for "file")</p> <p>group <math>\left\{ \begin{array}{l} (*abs) \\ (+rel) \\ (-rel) \end{array} \right\}</math></p> <p>(*abs) absolute generation number (1-9999); * and parentheses must be specified.</p> <p>(+rel) (-rel) relative generation number (0-99); sign and parentheses must be specified.</p>
integer	0...9, +, -	+ or -, if specified, must be the first character.
name	A...Z 0...9 \$, #, @	Must not begin with 0...9.
posix-filename	A...Z 0...9 special characters	String with a length of up to 255 characters; consists of either one or two periods or of alphanumeric characters and special characters. The special characters must be escaped with a preceding \ (backslash); the / is not allowed. Must be enclosed within single quotes if alternative data types are permitted, separators are used, or the first character is a ?, ! or ^. A distinction is made between uppercase and lowercase.

Table 2: Data types (part 3 of 5)



Data type	Character set	Special rules
vsn	a) A...Z 0...9  b) A...Z 0...9 \$, #, @	a) Input format: pvsid.sequence-no max. 6 characters pvsid: 2-4 characters; PUB must not be entered sequence-no: 1-3 characters  b) Max. 6 characters; PUB may be prefixed, but must not be followed by \$, #, @.
x-string	Hexadecimal: 00...FF	Must be enclosed in single quotes; must be prefixed by the letter X. There may be an odd number of characters.
x-text	Hexadecimal: 00...FF	Must not be enclosed in single quotes; the letter X must not be prefixed. There may be an odd number of characters.

Table 2: Data types (part 5 of 5)

### Special data types

Data type	Character set	Special rules
element-name	A...Z 0...9 \$, #, @ hyphen period underscore	The characters hyphen, underscore and period must not be the first or last character, and two of the same of any of these special characters must not immediately follow one another. The hyphen must not stand immediately to the right of any of the characters \$, @, #, under- score and period. The element name must contain at least one letter or one of the special characters \$, #, @.
element-version	A...Z 0...9 hyphen period	The special characters period and hyphen must not be the first or last character. Two of the same of these special characters must not immedi- ately follow one another. The hyphen must not stand immediately to the right of a period.
path-name	Any	See <a href="#">page 24</a> for input format.

Table 3: Special data types (part 1 of 2)

Data type	Character set	Special rules
symbol	A...Z 0...9 \$,#,@,&,% hyphen underscore	First character: A...Z or \$,#,@ In addition, names comprising 8 blanks are also permitted.
symbol-with-wild	EBCDIC characters	Must be enclosed in apostrophes; parts of a name may be replaced . by wildcard characters (see <a href="#">table 4</a> ).

Table 3: Special data types (part 2 of 2)



**Suffixes for data types**

<b>Suffix</b>	<b>Meaning</b>
	x     minimum value permitted for “integer”. x is an (optionally signed) integer.
	y     maximum value permitted for “integer”. y is an (optionally signed) integer.
	x     minimum length for the operand value; x is an integer.
	y     maximum length for the operand value; y is an integer.
	x=y   the length of the operand value must be precisely x.

Table 4: Data type suffixes (part 1 of 7)

Suffix	Meaning												
with (contd.)													
-wild(n)	<p>Parts of names may be replaced by the following wildcards. n denotes the maximum input length when using wildcards.</p> <p>Due to the introduction of the data types posix-filename and posix-pathname, SDF now accepts wildcards from the UNIX world (referred to below as POSIX wildcards) in addition to the usual BS2000 wildcards.</p> <p>However, as not all commands support POSIX wildcards, their use for data types other than posix-filename and posix-pathname can lead to semantic errors.</p> <p>Only POSIX wildcards or only BS2000 wildcards should be used within a search pattern. Only POSIX wildcards are allowed for the data types posix-filename and posix-pathname. If a pattern can be matched more than once in a string, the first match is used.</p> <table border="1"> <thead> <tr> <th>BS2000 wildcards</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>*</td> <td>Replaces an arbitrary (even empty) character string. If the string concerned starts with *, then the * must be entered twice in succession if it is followed by other characters and if the character string entered does not contain at least one other wildcard.</td> </tr> <tr> <td>Terminating period</td> <td>Partially-qualified entry of a name. Corresponds implicitly to the string <code>./**</code>, i.e. at least one other character follows the period.</td> </tr> <tr> <td>/</td> <td>Replaces any single character.</td> </tr> <tr> <td>&lt;s<sub>x</sub>:s<sub>y</sub>&gt;</td> <td>Replaces a string that meets the following conditions: <ul style="list-style-type: none"> <li>– It is at least as long as the shortest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It is not longer than the longest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It lies between s<sub>x</sub> and s<sub>y</sub> in the alphabetic collating sequence; numbers are sorted after letters (A...Z, 0...9)</li> <li>– s<sub>x</sub> can also be an empty string (which is in the first position in the alphabetic collating sequence)</li> <li>– s<sub>y</sub> can also be an empty string, which in this position stands for the string with the highest possible code (contains only the characters X'FF' )</li> </ul> </td> </tr> <tr> <td>&lt;s<sub>1</sub>,...&gt;</td> <td>Replaces all strings that match any of the character combinations specified by s. s may also be an empty string. Any such string may also be a range specification "s<sub>x</sub>:s<sub>y</sub>" (see above).</td> </tr> </tbody> </table>	BS2000 wildcards	Meaning	*	Replaces an arbitrary (even empty) character string. If the string concerned starts with *, then the * must be entered twice in succession if it is followed by other characters and if the character string entered does not contain at least one other wildcard.	Terminating period	Partially-qualified entry of a name. Corresponds implicitly to the string <code>./**</code> , i.e. at least one other character follows the period.	/	Replaces any single character.	<s <sub>x</sub> :s <sub>y</sub> >	Replaces a string that meets the following conditions: <ul style="list-style-type: none"> <li>– It is at least as long as the shortest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It is not longer than the longest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It lies between s<sub>x</sub> and s<sub>y</sub> in the alphabetic collating sequence; numbers are sorted after letters (A...Z, 0...9)</li> <li>– s<sub>x</sub> can also be an empty string (which is in the first position in the alphabetic collating sequence)</li> <li>– s<sub>y</sub> can also be an empty string, which in this position stands for the string with the highest possible code (contains only the characters X'FF' )</li> </ul>	<s <sub>1</sub> ,...>	Replaces all strings that match any of the character combinations specified by s. s may also be an empty string. Any such string may also be a range specification "s <sub>x</sub> :s <sub>y</sub> " (see above).
BS2000 wildcards	Meaning												
*	Replaces an arbitrary (even empty) character string. If the string concerned starts with *, then the * must be entered twice in succession if it is followed by other characters and if the character string entered does not contain at least one other wildcard.												
Terminating period	Partially-qualified entry of a name. Corresponds implicitly to the string <code>./**</code> , i.e. at least one other character follows the period.												
/	Replaces any single character.												
<s <sub>x</sub> :s <sub>y</sub> >	Replaces a string that meets the following conditions: <ul style="list-style-type: none"> <li>– It is at least as long as the shortest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It is not longer than the longest string (s<sub>x</sub> or s<sub>y</sub>)</li> <li>– It lies between s<sub>x</sub> and s<sub>y</sub> in the alphabetic collating sequence; numbers are sorted after letters (A...Z, 0...9)</li> <li>– s<sub>x</sub> can also be an empty string (which is in the first position in the alphabetic collating sequence)</li> <li>– s<sub>y</sub> can also be an empty string, which in this position stands for the string with the highest possible code (contains only the characters X'FF' )</li> </ul>												
<s <sub>1</sub> ,...>	Replaces all strings that match any of the character combinations specified by s. s may also be an empty string. Any such string may also be a range specification "s <sub>x</sub> :s <sub>y</sub> " (see above).												

Table 4: Data type suffixes (part 2 of 7)

Suffix	Meaning
with-wild(n) (contd.)	-s Replaces all strings that do not match the specified string s. The minus sign may only appear at the beginning of string s. Within the data types filename or partial-filename the negated string -s can be used exactly once, i.e. -s can replace one of the three name components: cat, user or file.
	Wildcards are not permitted in generation and version specifications for file names. Only system administration may use wildcards in user IDs. Wildcards cannot be used to replace the delimiters in name components cat (colon) and user (\$ and period).
POSIX wildcards	Meaning
*	Replaces any single string (including an empty string). An * appearing at the first position must be duplicated if it is followed by other characters and if the entered string does not include at least one further wildcard.
?	Replaces any single character; not permitted as the first character outside single quotes.
[c <sub>x</sub> -c <sub>y</sub> ]	Replaces any single character from the range defined by c <sub>x</sub> and c <sub>y</sub> , including the limits of the range. c <sub>x</sub> and c <sub>y</sub> must be normal characters.
[s]	Replaces exactly one character from string s. The expressions [c <sub>x</sub> -c <sub>y</sub> ] and [s] can be combined into [s <sub>1</sub> c <sub>x</sub> -c <sub>y</sub> s <sub>2</sub> ].
[!c <sub>x</sub> -c <sub>y</sub> ]	Replaces exactly one character not in the range defined by c <sub>x</sub> and c <sub>y</sub> , including the limits of the range. c <sub>x</sub> and c <sub>y</sub> must be normal characters. The expressions [!c <sub>x</sub> -c <sub>y</sub> ] and [s] can be combined into [!s <sub>1</sub> c <sub>x</sub> -c <sub>y</sub> s <sub>2</sub> ].
[!s]	Replaces exactly one character not contained in string s. The expressions [!s] and [!c <sub>x</sub> -c <sub>y</sub> ] can be combined into [!s <sub>1</sub> c <sub>x</sub> -c <sub>y</sub> s <sub>2</sub> ].

Table 4: Data type suffixes (part 3 of 7)

Suffix	Meaning										
with (contd.)											
wild-constr(n)	<p>Specification of a constructor (string) that defines how new names are to be constructed from a previously specified selector (i.e. a selection string with wildcards). See also with-wild. n denotes the maximum input length when using wildcards.</p> <p>The constructor may consist of constant strings and patterns. A pattern (character) is replaced by the string that was selected by the corresponding pattern in the selector.</p> <p>The following wildcards may be used in constructors:</p> <table border="1"> <thead> <tr> <th>Wildcard</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>*</td> <td>Corresponds to the string selected by the wildcard * in the selector.</td> </tr> <tr> <td>Terminating period</td> <td>Corresponds to the partially-qualified specification of a name in the selector; corresponds to the string selected by the terminating period in the selector.</td> </tr> <tr> <td>/ or ?</td> <td>Corresponds to the character selected by the / or ? wildcard in the selector.</td> </tr> <tr> <td>&lt;n&gt;</td> <td>Corresponds to the string selected by the n-th wildcard in the selector, where n is an integer.</td> </tr> </tbody> </table>	Wildcard	Meaning	*	Corresponds to the string selected by the wildcard * in the selector.	Terminating period	Corresponds to the partially-qualified specification of a name in the selector; corresponds to the string selected by the terminating period in the selector.	/ or ?	Corresponds to the character selected by the / or ? wildcard in the selector.	<n>	Corresponds to the string selected by the n-th wildcard in the selector, where n is an integer.
Wildcard	Meaning										
*	Corresponds to the string selected by the wildcard * in the selector.										
Terminating period	Corresponds to the partially-qualified specification of a name in the selector; corresponds to the string selected by the terminating period in the selector.										
/ or ?	Corresponds to the character selected by the / or ? wildcard in the selector.										
<n>	Corresponds to the string selected by the n-th wildcard in the selector, where n is an integer.										
	<p>Allocation of wildcards to corresponding wildcards in the selector:</p> <p>All wildcards in the selector are numbered from left to right in ascending order (global index).</p> <p>Identical wildcards in the selector are additionally numbered from left to right in ascending order (wildcard-specific index).</p> <p>Wildcards can be specified in the constructor by one of two mutually exclusive methods:</p> <ol style="list-style-type: none"> <li>1. Wildcards can be specified via the global index: &lt;n&gt;</li> <li>2. The same wildcard may be specified as in the selector; substitution occurs on the basis of the wildcard-specific index. For example: the second “/” corresponds to the string selected by the second “/” in the selector</li> </ol>										

Table 4: Data type suffixes (part 4 of 7)

Suffix	Meaning
with-wild-constr(n) (contd.)	<p>The following rules must be observed when specifying a constructor:</p> <ul style="list-style-type: none"> <li>– The constructor can only contain wildcards of the selector.</li> <li>– If the string selected by the wildcard &lt;...&gt; or [...] is to be used in the constructor, the index notation must be selected.</li> <li>– The index notation must be selected if the string identified by a wildcard in the selector is to be used more than once in the constructor. For example: if the selector “A/” is specified, the constructor “A&lt;n&gt;&lt;n&gt;” must be specified instead of “A//”.</li> <li>– The wildcard * can also be an empty string. Note that if multiple asterisks appear in sequence (even with further wildcards), only the last asterisk can be a non-empty string, e.g. for “*****” or “*//*”.</li> <li>– Valid names must be produced by the constructor. This must be taken into account when specifying both the constructor and the selector.</li> <li>– Depending on the constructor, identical names may be constructed from different names selected by the selector. For example: “A/*” selects the names “A1” and “A2”; the constructor “B*” generates the same new name “B” in both cases. To prevent this from occurring, all wildcards of the selector should be used at least once in the constructor.</li> <li>– If the constructor ends with a period, the selector must also end with a period. The string selected by the period at the end of the selector cannot be specified by the global index in the constructor specification.</li> </ul>

Table 4: Data type suffixes (part 5 of 7)

Suffix	Meaning																				
with-wild- constr(n) (contd.)	Examples:																				
	<table border="1"> <thead> <tr> <th>Selector</th> <th>Selection</th> <th>Constructor</th> <th>New name</th> </tr> </thead> <tbody> <tr> <td>A/*</td> <td>AB1 AB2 A.B.C</td> <td>D&lt;3&gt;&lt;2&gt;</td> <td>D1 D2 D.CB</td> </tr> <tr> <td>C.&lt;A:C&gt;/&lt;D,F&gt;</td> <td>C.AAD C.ABD C.BAF C.BBF</td> <td>G.&lt;1&gt;.&lt;3&gt;.XY&lt;2&gt;</td> <td>G.A.D.XYA G.A.D.XYB G.B.F.XYA G.B.F.XYB</td> </tr> <tr> <td>C.&lt;A:C&gt;/&lt;D,F&gt;</td> <td>C.AAD C.ABD C.BAF C.BBF</td> <td>G.&lt;1&gt;.&lt;2&gt;.XY&lt;2&gt;</td> <td>G.A.A.XYA G.A.B.XYB G.B.A.XYA G.B.B.XYB</td> </tr> <tr> <td>A/B</td> <td>ACDB ACEB AC.B A.CB</td> <td>G/XY/</td> <td>GCXYD GCXYE GCXY.<sup>1</sup> G.XYC</td> </tr> </tbody> </table>	Selector	Selection	Constructor	New name	A/*	AB1 AB2 A.B.C	D<3><2>	D1 D2 D.CB	C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<3>.XY<2>	G.A.D.XYA G.A.D.XYB G.B.F.XYA G.B.F.XYB	C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<2>.XY<2>	G.A.A.XYA G.A.B.XYB G.B.A.XYA G.B.B.XYB	A/B	ACDB ACEB AC.B A.CB	G/XY/	GCXYD GCXYE GCXY. <sup>1</sup> G.XYC
	Selector	Selection	Constructor	New name																	
	A/*	AB1 AB2 A.B.C	D<3><2>	D1 D2 D.CB																	
	C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<3>.XY<2>	G.A.D.XYA G.A.D.XYB G.B.F.XYA G.B.F.XYB																	
C.<A:C>/<D,F>	C.AAD C.ABD C.BAF C.BBF	G.<1>.<2>.XY<2>	G.A.A.XYA G.A.B.XYB G.B.A.XYA G.B.B.XYB																		
A/B	ACDB ACEB AC.B A.CB	G/XY/	GCXYD GCXYE GCXY. <sup>1</sup> G.XYC																		
<sup>1</sup> The period at the end of the name may violate naming conventions (e.g. for fully-qualified file names).																					
without	Restricts the specification options for a data type.																				
-cat	Specification of a catalog ID is not permitted.																				
-corr	Input format: [[C]][V][m]m.na['] Specifications for the data type product-version must not include the correction status.																				
-gen	Specification of a file generation or file generation group is not permitted.																				
-man	Input format: [[C]][V][m]m.n['] Specifications for the data type product-version must not include either release or correction status.																				
-odd	The data type x-text permits only an even number of characters.																				
-sep	With the data type "text", specification of the following separators is not permitted: ; = ( ) < > _ (i.e. semicolon, equals sign, left and right parentheses, greater than, less than, and blank).																				
-temp- file	Specification of a temporary file is not permitted (see #file or @file under filename).																				

Table 4: Data type suffixes (part 6 of 7)

Suffix	Meaning
without (contd.)	
-user	Specification of a user ID is not permitted.
-vers	Specification of the version (see “file(no)”) is not permitted for tape files.
-wild	The file types posix-filename and posix-pathname must not contain a pattern (character).
mandatory	Certain specifications are necessary for a data type.
-corr	Input format: <code>[[C]][V][m].na[so][ ]</code> Specifications for the data type product-version must include the correction status and therefore also the release status.
-man	Input format: <code>[[C]][V][m].na[so][ ]</code> Specifications for the data type product-version must include the release status. Specification of the correction status is optional if this is not prohibited by the use of the suffix without-corr.
-quotes	Specifications for the data types posix-filename and posix-pathname must be enclosed in single quotes.

Table 4: Data type suffixes (part 7 of 7)

## 7.3 Description of the statements

The statements are arranged in alphabetical order by name. The description of a statement is organized as follows:

- statement name
- description of the statement function
- statement format
- description of the statement operands.

**Overview**

<b>Statement name</b>	<b>Function</b>
BEGIN-SUB-LLM-STATEMENTS	Define beginning of a sub-LLM
END	Terminate BINDER run
END-SUB-LLM-STATEMENTS	Define end of a sub-LLM
INCLUDE-MODULES	Include modules
MERGE-MODULES	Merge the modules of a (sub-)LLM
MODIFY-ERROR-PROCESSING	Control error processing
MODIFY-LLM-ATTRIBUTES	Modify attributes of an LLM
MODIFY-MAP-DEFAULTS	Modify default values for the output of lists
MODIFY-MODULE-ATTRIBUTES	Modify the attributes of modules
MODIFY-STD-DEFAULTS	Modify the global default values
MODIFY-SYMBOL-ATTRIBUTES	Modify attributes of symbols
MODIFY-SYMBOL-TYPE	Modify symbol types
MODIFY-SYMBOL-VISIBILITY	Modify masking of symbols
REMOVE-MODULES	Remove modules
RENAME-SYMBOLS	Modify symbol names
REPLACE-MODULES	Replace modules
RESOLVE-BY-AUTOLINK	Resolve external references by autolink
SAVE-LLM	Save an LLM
SET-EXTERN-RESOLUTION	Handle unresolved external references
SET-USER-SLICE-POSITION	Define position of a slice
SHOW-DEFAULTS	Display the global default values
SHOW-LIBRARY-ELEMENTS	Display and check library elements
SHOW-MAP	Output lists
SHOW-SYMBOL-INFORMATION	Display symbol information
START-LLM-CREATION	Create an LLM
START-LLM-UPDATE	Update an LLM
START-STATEMENT-RECORDING	Records BINDER statements
STOP-STATEMENT-RECORDING	Terminates recording of BINDER statements

The SDF standard statements may also be specified in a BINDER-RUN. They (except for END) are not described in the present manual. A description of these statements can be found in the “Introductory Guide to the SDF Dialog Interface” [13].



## BEGIN-SUB-LLM-STATEMENTS

This statement is used for the logical structuring of an LLM. It defines the beginning of a sub-LLM within an LLM or sub-LLM.

Either the node at which the sub-LLM is to begin can be defined through the path name or the node of the current sub-LLM can be selected.

A sub-LLM is created in exactly the same way as an LLM, i.e. all statements are valid. The END-SUB-LLM-STATEMENTS statement terminates the structure of the sub-LLM.

### BEGIN-SUB-LLM-STATEMENTS

```

SUB-LLM-NAME = <c-string 1..32 with-low> / <text 1..32>
, PATH-NAME = *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
, RESOLUTION-SCOPE = *UNCHANGED / *STD / *PARAMETERS(...)
  *PARAMETERS(...)
    |
    | HIGH-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> /
    | <text 1..255>
    |
    | LOW-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> /
    | <text 1..255>
    |
    | FORBIDDEN-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> / <text 1..255>
    |
  , MODE = *CREATE / *UPDATE

```

### **SUB-LLM-NAME = <structured-name 1..32>**

Defines the name that the sub-LLM is to receive.

### **PATH-NAME =**

Defines the node in the logical structure at which the sub-LLM is to begin.

### **PATH-NAME = \*CURRENT-SUB-LLM**

The sub-LLM is to begin at the node of the current sub-LLM.

### **PATH-NAME = <text 1..255>**

Path name of the node at which the sub-LLM is to begin.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**RESOLUTION-SCOPE =**

Defines priority classes that control the order in which BINDER is to search other modules when resolving external references. Two values must be discriminated for each class:

- The dynamic value influences the order in which modules are searched to resolve external references. It is, however, not stored in the LLM.
- The static value is stored in the LLM and forms the basis for determining the dynamic value.

**RESOLUTION-SCOPE = \*UNCHANGED**

The static values of the priority classes are those stored in the modules concerned. They are assigned the value \*STD (see below) for object modules (OMs).

**RESOLUTION-SCOPE = \*STD**

The static values of the priority classes are \*STD, i.e. the dynamic values of superordinate nodes are taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static values of the ROOT node of an LLM is \*STD. The dynamic values in this case are \*NONE (see below).

**RESOLUTION-SCOPE = \*PARAMETERS(...)**

The static values of the individual priority classes are defined separately.

**HIGH-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is to be searched before all others for resolving external references (see [section “Rules for resolving external references” on page 81](#)).

**HIGH-PRIORITY-SCOPE = \*STD**

The static value of this priority class is \*STD, i.e. the dynamic value of the superordinate node is taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static value of the ROOT node of an LLM is \*STD. The dynamic value in this case is \*NONE (see below).

**HIGH-PRIORITY-SCOPE = \*NONE**

The HIGH-PRIORITY-SCOPE priority class is undefined, i.e. there are no modules which are to be searched before all others for resolving external references. The value of the superordinate node is not taken over.

**HIGH-PRIORITY-SCOPE = <c-string 1..255 with-low> / <text 1..255>**

Path name of the sub-LLM which is to be searched first for resolving external references.

**LOW-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is only to be searched for resolving external references after the search was unsuccessful in all other modules (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

**FORBIDDEN-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is not to be searched for resolving external references (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

An example of the inheritance mechanism for the PRIORITY-SCOPE values is provided in the description of the INCLUDE-MODULES statement on page [221](#).

**MODE =**

Specifies whether a new sub-LLM is created or an existing sub-LLM updated.

**MODE = \*CREATE**

A new sub-LLM is created.

**MODE = \*UPDATE**

An existing sub-LLM is updated.

## END

This statement terminates the BINDER run. The input sources opened by BINDER are closed and an end message is output.

<b>END</b>

The END statement has no operands.

## END-SUB-LLM-STATEMENTS

This statement is used for the logical structuring of a sub-LLM. It defines the end of a sub-LLM within an LLM or sub-LLM.

The current LLM is reset to the level that contained the current LLM prior to the associated BEGIN-SUB-LLM-STATEMENTS statement.

<b>END-SUB-LLM-STATEMENTS</b>

The END-SUB-LLM-STATEMENTS statement has no operands.

## INCLUDE-MODULES

This statement reads one or more modules from the specified input source and includes them in the current LLM work area.

Both object modules and LLMs can be included as modules. It is not possible to include LLMs with user-defined slices or LLMs without relocation information. If the entire structure information was included when the LLM was stored (operand LOGICAL-STRUCTURE = WHOLE-LLM), then either entire LLMs can be included or sub-LLMs can be selected.

The input source may be:

- for object modules: a program library (element type R), an object module library (OML) or the EAM object module file (OMF),
- for LLMs and sub-LLMs: a program library (element type L).

### INCLUDE-MODULES

**MODULE-CONTAINER** = **\*LIBRARY-ELEMENT** (...) / **\*FILE**(...) / **\*OMF**(...)

**\*LIBRARY-ELEMENT**(...)

**LIBRARY** = **\*CURRENT-INPUT-LIB** / <filename 1..54 without-gen-vers> / **\*LINK**(...) / **\*BLSLIB-LINK** / **\*OMF**

**\*LINK**(...)

| **LINK-NAME** = <structured-name 1..8> / <filename 1..8 without-gen>

, **ELEMENT** = **\*ALL** (...) / list-poss(40): <composed-name 1..64>(…) / <c-string 1..64>(…)

**\*ALL**(...)

| **VERSION** = **\*HIGHEST-EXISTING** / <composed-name 1..24> / <c-string 1..24>

<composed-name>(…)

| **VERSION** = **\*HIGHEST-EXISTING** / <composed-name 1..24> / <c-string 1..24>

, **SUB-LLM** = **\*WHOLE-LLM** / <c-string 1..255 with-low> / <text 1..255>

<c-string>(…)

| **VERSION** = **\*HIGHEST-EXISTING** / <composed-name 1..24> / <c-string 1..24>

, **SUB-LLM** = **\*WHOLE-LLM** / <c-string 1..255 with-low> / <text 1..255>

, **TYPE** = (**\*L**, **\*R**) / list-poss(2): **\*L** / **\*R**

continued ➔

```

*FILE(...)
  |
  | FILE-NAME = <filename 1..54 without-gen> / *LINK(...)
  |
  |   *LINK(...)
  |   |
  |   | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
  |   |
  |   | ,SUB-LLM = *WHOLE-LLM / <c-string 1..255 with-low> / <text 1..255>
  |
  | *OMF(...)
  | |
  | | ELEMENT = *ALL / list-poss(40): <composed-name 1..64> / <c-string 1..64>
  | |
  | | ,NAME = *INTERNAL / *ELEMENT-NAME / <c-string 1..32 with-low> / <text 1..32>
  | |
  | | ,PATH-NAME = *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
  | |
  | | ,SLICE = *CURRENT-SLICE / *ROOT / <structured-name 1..32>
  | |
  | | ,LOGICAL-STRUCTURE = *INCLUSION-DEFAULT / *WHOLE-LLM / *OBJECT-MODULES
  | |
  | | ,TEST-SUPPORT = *INCLUSION-DEFAULT / *NO / *YES
  | |
  | | ,RUN-TIME-VISIBILITY = *UNCHANGED / *NO / *YES
  | |
  | | ,RESOLUTION-SCOPE = *UNCHANGED / *STD / *PARAMETERS(...)
  | |
  | | *PARAMETERS(...)
  | | |
  | | | HIGH-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> /
  | | | <text 1..255>
  | | |
  | | | ,LOW-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> /
  | | | <text 1..255>
  | | |
  | | | ,FORBIDDEN-SCOPE = *UNCHANGED / *STD / *NONE / <c-string 1..255 with-low> / <text 1..255>
  | |
  | | ,NAME-COLLISION = *STD / *IGNORED / *WARNING(...) / *ERROR(...)
  | |
  | | *WARNING(...)
  | | |
  | | | SCOPE = *WHOLE-LLM / *SLICE
  | |
  | | *ERROR(...)
  | | |
  | | | SCOPE = *WHOLE-LLM / *SLICE

```

**MODULE-CONTAINER =**

Defines where the LLM is to be stored or is already stored.

**MODULE-CONTAINER = \*LIBRARY-ELEMENT(...)**

The LLM is stored in a program library.

**LIBRARY =**

Specifies the input source from which the modules are read.

**LIBRARY = \*CURRENT-INPUT-LIB**

The input source from which the *last* module (OM or LLM) was read by means of a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES statement is used. The scope of the operand relates to one edit run.

**LIBRARY = <filename 1..54 without-gen-vers>**

File name of the library that is to be used as the input source.

**LIBRARY = \*LINK(...)**

Denotes a library with the file link name

**LINKNAME = <structured-name 1..8>**

File link name of the library that is to be used as the input source.

**LIBRARY = \*BLSLIB-LINK**

The input sources are the libraries with the file link name BLSLIBnn (00≤nn≤99). The libraries are searched in *ascending* order of “nn” values for the file link name.

**LIBRARY = \*OMF**

The input source is the EAM object module file. This contains only object modules. (If the operand NAME has the value \*ELEMENT-NAME, the BINDER replaces this value with \*INTERNAL.)

**ELEMENT =**

Defines the element name and the element version of the modules that are included from the specified input source.

**ELEMENT = \*ALL(...)**

All modules are included from the specified input source.

**VERSION =**

Specifies the element version of the module. The element version is applicable only to program libraries.

**VERSION = \*HIGHEST-EXISTING**

BINDER takes as element version the default value for the highest version in the case of program libraries (see the “LMS” manual [3]).

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**ELEMENT = <composed-name 1..64>(…)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)).

**VERSION = \*HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>**

Specifies the element version of the module. The element version is applicable only to program libraries.

See above for meaning of operands.

**SUB-LLM =**

Specifies whether the whole LLM or a sub-LLM is included.

**SUB-LLM = \*WHOLE-LLM**

The whole LLM is included.

**SUB-LLM = <text 1..255>**

Path name of the sub-LLM that is included.

Note: BINDER checks special data type <path-name>.

**ELEMENT = <c-string 1..64>(…)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)).

See above for meaning of operands.

**TYPE =**

Defines the priority of the modules (object modules and/or LLMs) to be included.

**TYPE = (\*L,\*R)**

Both LLMs and object modules are included. If the same name is specified for an LLM as for an object module, the *LLM* is included.

**TYPE = (\*R,\*L)**

Both LLMs and object modules are included. If the same name is specified for an LLM as for an object module, the *object module* is included.

**TYPE = \*R**

Only object modules are included.

**TYPE = \*L**

Only LLMs are included.



**MODULE-CONTAINER = \*FILE(...)****FILE-NAME =**

The LLM is stored in a PAM file.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Name of the PAM file in which the LLM is stored.

**FILE-NAME = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the PAM file in which the LLM is stored.

**MODULE-CONTAINER = \*OMF(...)**

The input source is the EAM object module file. This contains only object modules. (If the NAME operand has the value \*ELEMENT-NAME, the BINDER replaces this value with \*INTERNAL in this case.)

**ELEMENT = \*ALL / list-poss(40): <composed-name 1..64> / <c-string 1..64>**

See above for the meanings of the operands.

**NAME =**

Specifies the logical name for the module to be included.

**NAME = \*INTERNAL**

The internal name is used as the logical name.

**NAME = \*ELEMENT-NAME**

The name of the library element is used as the logical name for the module. If necessary, the BINDER truncates this name to 32 characters.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**NAME = <structured-name 1..32>**

Explicit specification of the logical name.

**PATH-NAME =**

Defines the sub-LLM in the logical structure of the current LLM in the work area in which modules are included.

**PATH-NAME = \*CURRENT-SUB-LLM**

The current sub-LLM is assumed (see BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = <text 1..255>**

Path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**SLICE =**

Defines the slice in the physical structure of the LLM in which the modules are included. The slice must be defined with a SET-USER-SLICE-POSITION statement (see SET-USER-SLICE-POSITION statement).

**SLICE = \*CURRENT-SLICE**

Modules are included in the current slice. This is the slice that was defined by the most recent preceding SET-USER-SLICE-POSITION statement.

**SLICE = \*ROOT**

Modules are included in the root slice (%ROOT).

**SLICE = <structured-name 1..32>**

Explicit specification of the slice in which modules are included.

**LOGICAL-STRUCTURE =**

Specifies whether the logical structure information from the modules is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

The logical structure information is not taken over. A structure comprising only object modules (OMs) is set up in the current LLM.

**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is taken over into the current LLM.



Information on the existence of the list for symbolic debugging is provided in the "T&D" column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.

**RUN-TIME-VISIBILITY =**

Specifies whether the module is to be regarded as a runtime module. All symbols in a runtime module are masked when the module is stored and are, for the moment, not used for resolving external references. This masking of the symbols is canceled during any subsequent read access to the module (e.g. with START-LLM-UPDATE or INCLUDE-MODULES).

**RUN-TIME-VISIBILITY = \*UNCHANGED**

The value is not changed. When a module is included in an LLM for the first time with INCLUDE-MODULES or REPLACE-MODULES, BINDER assumes the value NO.

**RUN-TIME-VISIBILITY = \*NO**

The module is not to be regarded as a runtime module.

**RUN-TIME-VISIBILITY = \*YES**

The module is to be regarded as a runtime module. All symbols in the module are masked when the module is stored.

**RESOLUTION-SCOPE =**

Defines priority classes that control the order in which BINDER is to search other modules when resolving external references. Two values must be discriminated for each class:

- The dynamic value influences the order in which modules are searched to resolve external references. It is, however, not stored in the LLM.
- The static value is stored in the LLM and forms the basis for determining the dynamic value.

**RESOLUTION-SCOPE = \*UNCHANGED**

The static values of the priority classes are those stored in the modules concerned. They are assigned the value \*STD (see below) for object modules (OMs).

**RESOLUTION-SCOPE = \*STD**

The static values of the priority classes are \*STD, i.e. the dynamic values of superordinate nodes are taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static values of the ROOT node of an LLM is \*STD. The dynamic values in this case are \*NONE (see below).

**RESOLUTION-SCOPE = \*PARAMETERS(...)**

The static values of the individual priority classes are defined separately.

**HIGH-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is to be searched before all others for resolving external references (see [section "Rules for resolving external references" on page 81](#)).

**HIGH-PRIORITY-SCOPE = \*STD**

The static value of this priority class is \*STD, i.e. the dynamic value of the superordinate node is taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static value of the ROOT node of an LLM is \*STD. The dynamic value in this case is \*NONE (see below).

**HIGH-PRIORITY-SCOPE = \*NONE**

The HIGH-PRIORITY-SCOPE priority class is undefined, i.e. there are no modules which are to be searched before all others for resolving external references. The value of the superordinate node is not taken over.

**HIGH-PRIORITY-SCOPE = <c-string 1..255 with-low> / <text 1..255>**

Path name of the sub-LLM which is to be searched first for resolving external references.

**LOW-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is only to be searched for resolving external references after the search was unsuccessful in all other modules (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

**FORBIDDEN-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is not to be searched for resolving external references (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

The following example illustrates the inheritance mechanism for the PRIORITY-SCOPE values:

LLM1 and LLM2 are both in a library and static HIGH-PRIORITY-SCOPE values are defined for them as shown in the following table.

<b>Module</b>	<b>Dynamic value of HIGH-PRIORITY-SCOPE</b>
LLM1	LLM1.OM12
└─ OM11	*STD
└─ OM12	*NONE
LLM2	*STD
└─ OM21	LLM2.OM22
└─ OM22	*STD

LLM2 is linked into LLM1. During linkage, BINDER uses the static HIGH-PRIORITY-SCOPE values to determine the dynamic values which influence the search order when resolving external references.

The following table shows the resulting module structure and dynamic values of HIGH-PRIORITY-SCOPE.

<b>Module</b>	<b>Dynamic value of HIGH-PRIORITY-SCOPE</b>	<b>Comment</b>
LLM1	LLM1.OM12	= static value
└─ OM11	*STD	inherited from LLM1
└─ OM12	*NONE	= static value
└─ LLM2	*STD	inherited from LLM1
└─ OM21	LLM2.OM22	= static value
└─ OM22	*STD	inherited from LLM1

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## MERGE-MODULES

This statement merges all modules of an LLM or sub-LLM, creating a new LLM or sub-LLM which contains only one prelinked module with a single CSECT. The user can define the attributes for the new CSECT and specify which symbols are to remain in the External Symbols Vector.

### MERGE-MODULES

```

NAME = <c-string 1..32 with-low> / <text 1..32>
, PATH-NAME = *CURRENT-SUB-LLM / *NONE / <c-string 1..255 with-low> / <text 1..255>
, NEW-CSECT-NAME = *NAME / *STD / <c-string 1..32 with-low> / <text 1..32>
, NEW-CSECT-ATTRIBUTES = *PARAMETERS (...)
  *PARAMETERS(...)
    | RESIDENT = *STD / *YES / *NO / *UNIFORM
    | , PUBLIC = *STD / *NO / *YES / *UNIFORM
    | , READ-ONLY = *STD / *NO / *YES / *UNIFORM
    | , ALIGNMENT = *STD / *DOUBLE-WORD / *PAGE / *BUNDLE / <integer 3..12>
    | , ADDRESSING-MODE = *UNIFORM / *STD / *24 / *31 / *ANY
    | , RESIDENCY-MODE = *STD / *24 / *ANY / *UNIFORM
, ENTRY-LIST = *NONE / *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
, NAME-COLLISION = *STD / *IGNORED / *WARNING(...) / *ERROR(...)
  *WARNING(...)
    | SCOPE = *WHOLE-LLM / *SLICE
  *ERROR(...)
    | SCOPE = *WHOLE-LLM / *SLICE

```

**NAME** = <c-string 1..32 with-low> / <text 1..32>

The name of the sub-LLM or LLM to be merged.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**PATH-NAME** =

Specifies the next higher node in the logical structure of the sub-LLM to be merged.

**PATH-NAME** = \*CURRENT-SUB-LLM

The current sub-LLM is assumed

(see the BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = \*NONE**

The entire LLM which is currently being processed is to be merged.

**PATH-NAME = <text 1..255>**

The path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**NEW-CSECT-NAME =**

Specifies the name for the merged CSECT.

**NEW-CSECT-NAME = \*NAME**

The new CSECT receives the name of the merged object module.

**NEW-CSECT-NAME = \*STD**

The new CSECT receives the name of the first CSECT found in the (sub-)LLM to be merged.

**NEW-CSECT-NAME = <text 1..32>**

Explicit specification of the new CSECT name.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**NEW-CSECT-ATTRIBUTES = \*PARAMETERS(...)**

Defines the attributes for the new CSECT.

**RESIDENT =**

Defines the value for the attribute RESIDENT.

**RESIDENT = \*STD**

The new CSECT receives the attribute RESIDENT if *at least one* CSECT with the attribute RESIDENT exists in the sub-LLM to be merged; otherwise, the new CSECT is pageable (RESIDENT=NO).

**RESIDENT = \*YES**

The new CSECT is resident.

**RESIDENT = \*NO**

The new CSECT is pageable.

**RESIDENT = \*UNIFORM**

For all CSECTs included in the merge operation, the value of the attribute RESIDENT must be the same. If not, the MERGE-MODULES statement is rejected. The new CSECT also receives this attribute value.



**PUBLIC =**

Defines the value for the attribute PUBLIC.

**PUBLIC = \*STD**

The new CSECT is PRIVATE (PUBLIC=NO) if *at least one* CSECT with the attribute PUBLIC=NO exists in the sub-LLM to be merged; otherwise, the new CSECT is shareable (PUBLIC=YES).

**PUBLIC = \*NO**

The new CSECT is not shareable (PRIVATE).

**PUBLIC = \*YES**

The new CSECT is shareable.

**PUBLIC = \*UNIFORM**

For all CSECTs included in the merge operation, the value of the attribute PUBLIC must be the same. If not, the MERGE-MODULES statement is rejected. The new CSECT also receives this attribute value.

**READ-ONLY =**

Defines the value for the attribute READ-ONLY.

**READ-ONLY = \*STD**

Read/write access is permitted for the new CSECT if *at least one* CSECT with the attribute READ-ONLY=NO exists in the sub-LLM to be merged; otherwise, only read access is permitted for the new CSECT (READ-ONLY=YES).

**READ-ONLY = \*NO**

Only read access is permitted for the new CSECT.

**READ-ONLY = \*YES**

Read and write access is permitted for the new CSECT.

**READ-ONLY = \*UNIFORM**

For all CSECTs included in the merge operation, the value of the attribute READ-ONLY must be the same. If not, the MERGE-MODULES statement is rejected. The new CSECT also receives this attribute value.

**ALIGNMENT =**

Defines the alignment of the new CSECT.

**ALIGNMENT = \*STD**

The alignment of the new CSECT is the *largest* alignment of all CSECTs to be merged.

**ALIGNMENT = \*DOUBLE-WORD**

The new CSECT is aligned on a doubleword boundary.

**ALIGNMENT = \*PAGE**

The new CSECT is aligned on a page boundary, i.e. the address is a multiple of 4096 (X'1000').

**ALIGNMENT = \*BUNDLE**

The new CSECT is aligned on an address that is a multiple of 16.

**ALIGNMENT = <integer 3..12>**

The new CSECT is aligned on an address which is a multiple of  $2^n$ . The exponent “n” is specified with <integer 3..12>. (For example,  $2^3$  means alignment on a doubleword boundary and  $2^{12}$  means alignment on a page boundary.)

**ADDRESSING-MODE =**

Defines the addressing mode (AMODE) for the new CSECT.

**ADDRESSING-MODE = \*UNIFORM**

If the attribute AMODE is not the same for all CSECTs included in the merge operation, the statement is rejected. The new CSECT also receives this attribute value.

**ADDRESSING-MODE = \*STD**

The addressing mode for the new CSECT is the same as that for the first CSECT in the sub-LLM to be merged.

**ADDRESSING-MODE = 24**

The new CSECT is assigned 24-bit addressing mode.

**ADDRESSING-MODE = 31**

The new CSECT is assigned 31-bit addressing mode.

**ADDRESSING-MODE = \*ANY**

The new CSECT is assigned 24-bit or 31-bit addressing mode. The decision as to which addressing mode is to be used is made only when the CSECT is loaded.

**RESIDENCY-MODE =**

Defines the residency mode for the new CSECT.

**RESIDENCY-MODE = \*STD**

The residency mode for the new CSECT is 24 if at least one of the CSECTs to be merged has residency mode 24; otherwise, the value ANY is assumed.

**RESIDENCY-MODE = 24**

The residency mode for the new CSECT is 24.

**RESIDENCY-MODE = \*ANY**

The new CSECT may be loaded either above or below 16 Mbytes.

**RESIDENCY-MODE = \*UNIFORM**

For all CSECTs included in the merge operation, the value of the attribute RESIDENCY-MODE must be the same. If not, the MERGE-MODULES statement is rejected. The new CSECT also receives this attribute value.

**ENTRY-LIST =**

Defines the symbols (CSECTs or ENTRYs) which are to remain in the External Symbols Vector (ESV). Any CSECTs which are retained are converted into ENTRYs.

**ENTRY-LIST = \*NONE**

No symbols are to remain in the ESV.

**ENTRY-LIST = \*ALL**

All symbols are to remain in the ESV.

**ENTRY-LIST = <c-string 1..255>**

All symbols which match the wildcard pattern are to remain in the ESV.

Note: BINDER checks special data type <symbol-with-wild> (see [page 199](#)).

**ENTRY-LIST = <text 1..32>**

Explicit specification of the symbols which are to remain in the ESV.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled. A name conflict can occur only if a CSECT receives a new name.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## MODIFY-ERROR-PROCESSING

This statement controls error processing and termination of the BINDER run.

The following can be defined:

- the lowest severity class as of which messages are output,
- the output medium used for output of the messages,
- the severity class as of which the BINDER run is to be terminated and
- whether user switches and/or task switches are set on the occurrence of certain errors.

The scope of all operands relates to one BINDER run. The value **\*UNCHANGED** in the operands means that the default value is retained.

In the first MODIFY-ERROR-PROCESSING statement, the *first* operand value following the value UNCHANGED is assumed for UNCHANGED.

### MODIFY-ERROR-PROCESSING

**MESSAGE-CONTROL** = \*UNCHANGED / \*INFORMATION / \*WARNING / \*ERROR

,**MESSAGE-DESTINATION** = \*UNCHANGED / \*SYSOUT / \*SYSLST / \*BOTH

,**MAX-ERROR-WEIGHT** = \*UNCHANGED / \*FATAL / \*RECOVERABLE / \*SYNTAX /  
\*UNRESOLVED-EXTERNS / \*WARNING

,**SPECIAL-HANDLING** = \*UNCHANGED / \*NO / \*ALL(...) / \*PARAMETERS(...)

\*ALL(...)

**USER-SWITCH** = \*UNCHANGED / \*NO / <integer 0..31>

    ,**TASK-SWITCH** = \*UNCHANGED / \*NO / <integer 0..31>

continued ➡

```

*PARAMETERS(...)
  WARNING = *UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>
    ,UNRESOLVED-EXTERNS = *UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>
    ,SYNTAX-ERROR = *UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>
    ,RECOVERABLE-ERROR = *UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>
    ,FATAL-ERROR = UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>
    ,INTERNAL-ERROR = *UNCHANGED / *NO / *PARAMETERS(...)
    *PARAMETERS(...)
      | USER-SWITCH = *UNCHANGED / *NO / <integer 0..31>
      | ,TASK-SWITCH = *UNCHANGED / *NO / <integer 0..31>

```

**MESSAGE-CONTROL = \*UNCHANGED / \*INFORMATION / \*WARNING / \*ERROR**

Defines the lowest severity class as of which messages are output (see [page 134](#)).

**MESSAGE-CONTROL = \*INFORMATION**

The messages of all severity classes are output.

**MESSAGE-CONTROL = \*WARNING**

Only messages as of severity class WARNING are output. Messages of severity class INFORMATION are not output.

**MESSAGE-CONTROL = \*ERROR**

Only messages of severity classes SYNTAX ERROR, RECOVERABLE ERROR, FATAL ERROR and INTERNAL ERROR are output. Messages of severity classes INFORMATION, WARNING and UNRESOLVED EXTERNS are not output.

**MESSAGE-DESTINATION = \*UNCHANGED / \*SYSOUT / \*SYSLST / \*BOTH**

Defines the output destination for the messages.

**MESSAGE-DESTINATION = \*SYSOUT**

System file SYSOUT is the output destination.

**MESSAGE-DESTINATION = \*SYSLST**

System file SYSLST is the output destination.

**MESSAGE-DESTINATION = \*BOTH**

System files SYSOUT and SYSLST are the output destination.

**MAX-ERROR-WEIGHT = \*UNCHANGED / \*FATAL / \*RECOVERABLE / \*SYNTAX / \*UNRESOLVED-EXTERNS / \*WARNING**

Specifies which severity class will terminate the BINDER run. The BINDER run will be terminated upon the occurrence of all errors whose severity class is equal to or greater than the specified severity class (see [page 134](#)).

**SPECIAL-HANDLING = \*UNCHANGED / \*NO / \*ALL(...) / \*PARAMETERS(...)**

Specifies whether user or task switches are to be set upon the occurrence of errors. If so, it is possible to differentiate as to whether the specified user or task switch is set

- upon the occurrence of any error or
- upon the occurrence an error of a specific severity class.

**SPECIAL-HANDLING = \*NO**

No switches are set.

**SPECIAL-HANDLING = \*ALL(...)**

All errors result in setting of the specified user or task switch. Only one switch may be set for each severity class.

**USER-SWITCH = \*UNCHANGED / \*NO / <integer 0..31>**

Number of the user switch.

With NO, no user switch is set.

**TASK-SWITCH = \*UNCHANGED / \*NO / <integer 0..31>**

Number of the task switch.

With NO, no task switch is set.

**SPECIAL-HANDLING = \*PARAMETERS(...)**

Upon the occurrence of errors of the selected severity classes, the specified user or task switches are set. Only one switch may be set for each severity class.

**WARNING = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class WARNING.

**WARNING = \*NO**

No switch is set.

**WARNING = \*PARAMETERS (...)**

Defines the number of the switch that is set.

**USER-SWITCH = \*UNCHANGED / \*NO / <integer 0..31>**

Number of the user switch.

With NO, no user switch is set.

**TASK-SWITCH = \*UNCHANGED / \*NO / <integer 0..31>**

Number of the task switch.

With NO, no task switch is set.

**UNRESOLVED-EXTERNS = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class UNRESOLVED EXTERNS.

See WARNING for operands.

**SYNTAX-ERROR = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class SYNTAX ERROR.

See WARNING for operands.

**RECOVERABLE-ERROR = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class RECOVERABLE ERROR.

See WARNING for operands.

**FATAL-ERROR = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class FATAL ERROR.

See WARNING for operands.

**INTERNAL-ERROR = \*UNCHANGED / \*NO / \*PARAMETERS(...)**

Specifies whether a switch is set for the severity class INTERNAL ERROR. See

WARNING for operands.



## MODIFY-LLM-ATTRIBUTES

This statement modifies attributes of an LLM that were defined by means of the START-LLM-CREATION or START-LLM-UPDATE statements.

The following attributes can be modified:

- the internal name (INTERNAL-NAME)
- the internal version (INTERNAL-VERSION)
- the physical structure of the LLM (SLICE-DEFINITION)
- the copyright information (COPYRIGHT)
- use of the logical structure information and LSD information (INCLUSION-DEFAULTS).

The type of physical structure of the LLM may be modified as follows:

1. LLM with slices by attributes → LLM with single slice
2. LLM with single slice → LLM with slices by attributes
3. LLM with slices by attributes → LLM with slices by *other* attributes
4. LLM with user-defined slices → LLM with user-defined slices and modified values for AUTOMATIC-CONTROL and EXCLUSIVE-SLICE-CALL.

The default value **\*UNCHANGED** in the relevant operands means that the previously declared value applies in each case.

## MODIFY-LLM-ATTRIBUTES

```

INTERNAL-NAME = *UNCHANGED / <c-string 1..32 with-low> / <text 1..32>
, INTERNAL-VERSION = *UNCHANGED / *UNDEFINED / <composed-name 1..24> / <c-string 1..24>
, SLICE-DEFINITION = *UNCHANGED / *SINGLE / *BY-ATTRIBUTES(...) / *BY-USER(...)
  *BY-ATTRIBUTES(...)
    | READ-ONLY = *UNCHANGED / *NO / *YES
    | , RESIDENT = *UNCHANGED / *NO / *YES
    | , PUBLIC = *UNCHANGED / *NO / *YES(...)
    |   *YES(...)
    |     | SUBSYSTEM-ENTRIES = *NONE / list-poss(40): <c-string 1..32 with-low> /
    |       <text 1..32>
    |   , RESIDENCY-MODE = *UNCHANGED / *NO / *YES
  *BY-USER(...)
    | AUTOMATIC-CONTROL = *UNCHANGED / *NO / *YES(...)
    |   *YES(...)
    |     | RELOAD-SLICE= *YES / *NO
    |   , EXCLUSIVE-SLICE-CALL = *UNCHANGED / *NO / *YES
, COPYRIGHT = *UNCHANGED / *PARAMETERS(...) / *NONE
  *PARAMETERS(...)
    | NAME = *UNCHANGED / *SYSTEM-DEFAULT / <c-string 1..64 with-low>
    | , YEAR = *UNCHANGED / *CURRENT / <integer 1900..2100>
    | , PATH-NAME = *UNCHANGED / *NONE / <c-string 1..255 with-low> / <text 1..255>
    | , ENTRY = *UNCHANGED / *NONE / <c-string 1..32 with-low> / <text 1..32>
, INCLUSION-DEFAULTS = *PARAMETERS (...)
  *PARAMETERS(...)
    | LOGICAL-STRUCTURE = *UNCHANGED / *WHOLE-LLM / *OBJECT-MODULES
    | , TEST-SUPPORT = *UNCHANGED / *NO / *YES

```

**INTERNAL-NAME** = \*UNCHANGED / <structured-name 1..32>

Defines the new internal name of the LLM.

**INTERNAL-VERSION = \*UNCHANGED / \*UNDEFINED / <composed-name 1..24> / <c-string 1..24>**

Defines the new internal version of the LLM.

**INTERNAL-VERSION = \*UNDEFINED**

When the LLM is saved with the SAVE-LLM statement, the default value for the highest version for program libraries is assumed (see the “LMS” manual [3]).

**INTERNAL-VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the new internal version of the LLM.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**SLICE-DEFINITION = \*UNCHANGED / \*SINGLE / \*BY-ATTRIBUTES(...) / \*BY-USER(...)**

Defines the new physical structure of the LLM.

**SLICE-DEFINITION = \*SINGLE**

The LLM consists of a single slice.

**SLICE-DEFINITION = \*BY-ATTRIBUTES(...)**

The LLM consists of slices formed by the combination of the attributes of CSECTs (see [page 18ff](#)). If the BY-ATTRIBUTES operand is specified and if all sub-operands are set to NO, SINGLE is assumed. The operand values of READ-ONLY, RESIDENT, PUBLIC and RESIDENCY-MODE control only the assembly of slices. They have no effect on the individual CSECTs. Up to 16 different slices can be formed by the combination of the attributes.

**READ-ONLY = \*UNCHANGED / \*NO / \*YES**

Specifies whether the attribute READ-ONLY is to be considered when forming the slices.

If YES is specified, BINDER generates separate slices for CSECTs with different values for the attribute READ-ONLY.

**RESIDENT = \*UNCHANGED / \*NO / \*YES**

Specifies whether the attribute RESIDENT is to be considered when forming the slices. If YES is specified, BINDER generates separate slices for CSECTs with different values for the attribute RESIDENT.

**PUBLIC = \*UNCHANGED / \*NO / \*YES(...)**

Specifies whether the attribute PUBLIC is to be considered when forming the slices.

**PUBLIC = \*NO**

The attribute PUBLIC is not considered when forming the slices.

**PUBLIC = \*YES(...)**

BINDER generates separate slices for CSECTs with different values for the attribute PUBLIC.

**SUBSYSTEM-ENTRIES =**

Specifies the symbols (CSECT or ENTRY) of the PUBLIC slice which can be used for resolving external references if the PUBLIC slice is loaded as a dynamic subsystem (see the “Introductory Guide to Systems Support” [9]).

**SUBSYSTEM-ENTRIES = \*NONE**

No symbols from this subsystem (of the PUBLIC slice) are used for resolving external references.

**SUBSYSTEM-ENTRIES = <text 1..32>**

The name of the CSECT or the ENTRY in the PUBLIC slice loaded as a subsystem which can be used for resolving external references.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**RESIDENCY-MODE = \*UNCHANGED / \*NO / \*YES**

Specifies whether the attribute RMODE is to be considered when forming the slices. If YES is specified, BINDER generates separate slices for CSECTs with different values for the attribute RMODE.

**SLICE-DEFINITION = \*BY-USER(...)**

The physical structure of the LLM is defined by the user with SET-USER-SLICE-POSITION statements (user-defined slices). When this is done, overlays can be defined.

**AUTOMATIC-CONTROL = \*UNCHANGED / \*NO / \*YES(...)**

This is relevant only for overlays.

Specifies whether an Overlay Control Module (OCM) is linked into the created LLM in order to control automatic dynamic loading of the overlays.

**AUTOMATIC-CONTROL = \*YES(...)**

An OCM is linked into the created LLM to control automatic dynamic loading of overlays.

**RELOAD-SLICE=**

Specifies whether a loaded slice must be reloaded overwriting the previous slice in main memory.

**RELOAD-SLICE= \*YES**

An already loaded slice is reloaded overwriting the previous slice in main memory.

**RELOAD-SLICE= \*NO**

An already loaded slice is kept unchanged in main memory.

**EXCLUSIVE-SLICE-CALL = \*UNCHANGED / \*NO / \*YES**

This is relevant only for overlays and specifies whether external references between exclusive slices are to be resolved.

**EXCLUSIVE-SLICE-CALL = \*NO**

Specifies that BINDER is simply to report external references and is not to resolve any references between exclusive slices it may detect.

**EXCLUSIVE-SLICE-CALL = \*YES**

Specifies that BINDER is to resolve external references between exclusive slices, i.e. the user is willing to accept any errors which may possibly occur as a result of this.

**COPYRIGHT = \*UNCHANGED / \*PARAMETERS (...) / \*NONE**

Defines the new copyright information. The copyright information consists of text and the year number.

**COPYRIGHT = \*PARAMETERS(...)****NAME = \*UNCHANGED / \*SYSTEM-DEFAULT / <c-string 1..64>**

Denotes the new text for the copyright information.

**NAME = \*SYSTEM-DEFAULT**

The value of the class 2 system parameter BLSCOPYN is to be taken over. This value is defined at system installation time (see the "Introductory Guide to Systems Support" [9]).

**NAME = <c-string 1..64>**

New text for the copyright information. If the text comprises blanks, no copyright information is taken over.

**YEAR = \*UNCHANGED / \*CURRENT / <integer 1900..2100>**

Denotes the year number.

**YEAR = \*CURRENT**

Current year number.

**YEAR = <integer 1900..2100>**

Explicit specification of the year number.

**COPYRIGHT = \*NONE**

No new copyright information is taken over.

**INCLUSION-DEFAULTS =**

Defines the use of the logical structure information and LSD information. This is the default value that is used in the INCLUDE-MODULES, REPLACE-MODULES and RESOLVE-BY-AUTOLINK statements of the same edit run if no specific values are specified in these statements.

Logical structure information and LSD information are not included during saving of the LLM unless this is required both in the SAVE-LLM statement and in preceding INCLUDE-MODULES, REPLACE-MODULES or RESOLVE-BY-AUTOLINK statements.

**INCLUSION-DEFAULTS = \*PARAMETERS(...)****LOGICAL-STRUCTURE = \*UNCHANGED / \*WHOLE-LLM / \*OBJECT-MODULES**

Specifies whether the logical structure information is taken over from the modules into the current LLM when including or replacing modules.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

The logical structure information is not taken over. A structure comprising only object modules (OMs) is set up in the current LLM.

**TEST-SUPPORT = \*UNCHANGED / \*NO / \*YES**

Specifies whether the LSD information from the modules is taken over into the current LLM when including or replacing modules.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.

## MODIFY-MAP-DEFAULTS

This statement modifies the default values for a subsequent SHOW-MAP statement and for the next sequential SAVE-LLM statement with the MAP=YES operand.

The scope of all operands relates to one *BINDER run*.

A new MODIFY-MAP-DEFAULTS statement overwrites the operand values of the preceding MODIFY-MAP-DEFAULTS statement.

Since the MAP-NAME operand permits the user to specify names for lists, it is possible to define several different lists with different names. When the SHOW-MAP statement is executed in a BINDER run, the user can access the predefined lists by means of these names.

The value **\*UNCHANGED** in the relevant operands means that the value specified for this operand in the preceding MODIFY-MAP-DEFAULTS statement within a BINDER run is to be used.

In the first MODIFY-MAP-DEFAULTS statement, the *first* operand value following the value \*UNCHANGED is assumed for \*UNCHANGED.

### MODIFY-MAP-DEFAULTS

```

MAP-NAME = *STD / <structured-name 1..32>
,USER-COMMENT = *UNCHANGED / *NONE / <c-string 1..255 with-low>
,HELP-INFORMATION = *UNCHANGED / *YES / *NO
,GLOBAL-INFORMATION = *UNCHANGED / *YES / *NO
,LOGICAL-STRUCTURE = *UNCHANGED / *YES(...) / *NO
  *YES(...)
    | RESOLUTION-SCOPE = *UNCHANGED / *YES / *NO
    | ,HSI-CODE = *UNCHANGED / *YES / *NO / *X86
,PHYSICAL-STRUCTURE = *UNCHANGED / *YES / *NO

```

continued ➡

```

,PROGRAM-MAP = *UNCHANGED / *PARAMETERS(...) / *NO
  *PARAMETERS(...)
    |
    | DEFINITIONS = *UNCHANGED / *ALL / *NONE / list-poss(5): *MODULE / *CSECT /
    | *ENTRY / *COMMON / *XDSECT-D
    | ,INVERTED-XREF-LIST = *UNCHANGED / *NONE / *ALL / list-poss(4): *EXTRN / *VCON /
    | *WXTRN / *XDSECT-R
    | ,REFERENCES = *UNCHANGED / *ALL / *NONE / list-poss(4): *EXTRN / *VCON / *WXTRN /
    | *XDSECT-R
,UNRESOLVED-LIST = *UNCHANGED / *SORTED(...) / *YES(...) / *NO
  *SORTED(...)
    |
    | WXTRN = *YES / *NO
    | ,NOREF = *NO / *YES
  *YES(...)
    |
    | WXTRN = *YES / *NO
    | ,NOREF = *NO / *YES
,SORTED-PROGRAM-MAP = *UNCHANGED / *NO / *YES
,PSEUDO-REGISTER = *UNCHANGED / *NO / *YES
,UNUSED-MODULE-LIST = *UNCHANGED / *NO / *YES
,DUPLICATE-LIST = *UNCHANGED / *NO / *YES(...)
  *YES(...)
    |
    | INVERTED-XREF-LIST = *YES / *NO
,MERGED-MODULES = *UNCHANGED / *YES / *NO
,INPUT-INFORMATION = *UNCHANGED / *YES / *NO
,STATEMENT-LIST = *UNCHANGED / *NO / *YES
,OUTPUT = *UNCHANGED / *SYSLST(...) / *BY-SHOW-FILE(...) / <filename 1..54 without-gen-vers>(...) /
  *LINK(...) / *EXIT-ROUTINE(...)
  *SYSLST(...)
    |
    | SYSLST-NUMBER = *STD / <integer 1..99>
    | ,LINES-PER-PAGE = 64 / <integer 10..2147483647> / *IGNORED
    | ,LINE-SIZE = 136 / <integer 132..255>

```

continued ➡



```

*BY-SHOW-FILE(...)
  | FILE-NAME = *STD / <filename 1..54 without-gen-vers>
  | ,DELETE-FILE = *YES / *NO
  | ,LINE-SIZE = 136 / <integer 132..255>
<filename>(...)
  | LINE-SIZE = 136 / <integer 132..255>
*LINK(...)
  | LINK-NAME = BNDMAP / <structured-name 1..8> / <filename 1..8 without-gen>
  | ,LINE-SIZE = 136 / <integer 132..255>
*EXIT-ROUTINE(...)
  | ROUTINE-NAME = <c-string 1..32 with-low> / <text 1..32>
  | ,LIBRARY = *BLSLIB-LINK / <filename 1..54 without-gen-vers> / *LINK(...)
    | *LINK(...)
      | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
    | ,FILE-NAME = *STD / <filename 1..54 without-gen-vers>
    | ,LINE-SIZE = 136 / <integer 132..255>
    | ,USER-PARAMETERS = *NONE / <c-string 1..255 with-low> / <text 1..255>

```

**MAP-NAME =**

Specifies the name of the list to which the new default values are to apply.

**MAP-NAME = \*STD**

Applies the values to the default list with the name `BNDMAP.date.time.<tsn>`, i.e. to the list which is output for `//SHOW-MAP MAP-NAME=*STD` or for `//SAVE-LLM ... MAP=YES`.

**MAP-NAME = <structured-name 1..32>**

Applies the default values to a list defined and named by the user, which can be output with `//SHOW-MAP MAP-NAME=<structured-name 1..32>`.

See the SHOW-MAP statement, [page 312ff](#), for the meanings of the other operands.

## MODIFY-MODULE-ATTRIBUTES

This statement permits the user to modify the logical structure of an LLM, the runtime information and the list for symbolic debugging of a module.

### MODIFY-MODULE-ATTRIBUTES

```

NAME = <c-string 1..32 with-low> / <text 1..32>
, PATH-NAME = *CURRENT-SUB-LLM / *NONE / <c-string 1..255 with-low> / <text 1..255>
, NEW-NAME = *UNCHANGED / *INTERNAL / *ELEMENT-NAME / <c-string 1..32 with-low> / <text 1..32>
, NEW-PATH-NAME = *UNCHANGED / *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
, TEST-SUPPORT = *UNCHANGED / *INCLUSION-DEFAULT / *NO / *YES
, RUN-TIME-VISIBILITY = *UNCHANGED / *NO / *YES
, RESOLUTION-SCOPE = *UNCHANGED / *STD / *PARAMETERS(...)
  *PARAMETERS(...)
    | HIGH-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>
    | , LOW-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>
    | , FORBIDDEN-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>
, NAME-COLLISION = *STD / *IGNORED / *WARNING(...) / *ERROR(...)
  *WARNING(...)
    | SCOPE = *WHOLE-LLM / *SLICE
  *ERROR(...)
    | SCOPE = *WHOLE-LLM / *SLICE

```

**NAME = <structured-name 1..32> / <text 1..32>**

Specifies the name of the sub-LLM which is to be modified.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**PATH-NAME =**

Specifies the logically next higher node of the sub-LLM to be processed. By default, this is the current sub-LLM in the BINDER work area (see the BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = \*CURRENT-SUB-LLM**

The current sub-LLM is assumed

(see the BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = \*NONE**

The entire LLM is to be modified.

**PATH-NAME = <text 1..255>**

The path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**NEW-NAME =**

Specifies the new name for the (sub-)LLM.

**NEW-NAME = \*UNCHANGED**

The name of the (sub-)LLM remains unchanged.

**NEW-NAME = \*INTERNAL**

The internal name of the (sub-)LLM is used as the new name.

**NEW-NAME = \*ELEMENT-NAME**

The name which the sub-LLM possesses as a library element name is used as the new name. This value is permitted only if the module actually exists as a library element.

**NEW-NAME = <structured-name 1..32>**

Explicit specification of the new name for the (sub-)LLM.

**NEW-PATH-NAME =**

Specifies the new path name for the sub-LLM, thus permitting the logical structure of the LLM to be modified.

**NEW-PATH-NAME = \*UNCHANGED**

The path name of the sub-LLM remains unchanged, which means that the logical structure of the LLM also remains unchanged.

**NEW-PATH-NAME = \*CURRENT-SUB-LLM**

The path name of the current sub-LLM is used as the new path name (see [page 18](#)).

**NEW-PATH-NAME = <text 1..255>**

Explicit specification of the new path name.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is to be transferred into the current LLM.



Information on the existence of the list for symbolic debugging is provided in the “T&D” column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*UNCHANGED**

The LSD information remains unchanged.

**TEST-SUPPORT = \*INCLUSION-DEFAULT**

The values specified for the INCLUSION-DEFAULTS operand in the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statement in the same edit run are assumed.

**TEST-SUPPORT = \*NO**

The LSD information is not transferred.

**TEST-SUPPORT = \*YES**

The LSD information is transferred.

**RUN-TIME-VISIBILITY =**

Specifies whether the module is to be regarded as a runtime module. All symbols in a runtime module are masked when the module is stored and are, for the moment, not used for resolving external references. This masking of the symbols is canceled during any subsequent read access to the module (e.g. with START-LLM-UPDATE or INCLUDE-MODULES).

**RUN-TIME-VISIBILITY = \*UNCHANGED**

The value is not changed.

**RUN-TIME-VISIBILITY = \*NO**

The module is not to be regarded as a runtime module.

**RUN-TIME-VISIBILITY = \*YES**

The module is to be regarded as a runtime module. All symbols in the module are masked when the module is stored.

**RESOLUTION-SCOPE =**

Defines priority classes that control the order in which BINDER is to search other modules when resolving external references. Two values must be discriminated for each class:

- The dynamic value  
influences the order in which modules are searched to resolve external references. It is, however, not stored in the LLM.
- The static value  
is stored in the LLM and forms the basis for determining the dynamic value.

**RESOLUTION-SCOPE = \*UNCHANGED**

The static values of the priority classes are those stored in the modules concerned. They are assigned the value \*STD (see below) for object modules (OMs).

**RESOLUTION-SCOPE = \*STD**

The static values of the priority classes are \*STD, i.e. the dynamic values of superordinate nodes are taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static values of the ROOT node of an LLM is \*STD. The dynamic values in this case are \*NONE (see below).

**RESOLUTION-SCOPE = \*PARAMETERS(...)**

The static values of the individual priority classes are defined separately.

**HIGH-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is to be searched before all others for resolving external references (see [section "Rules for resolving external references" on page 81](#)).

**HIGH-PRIORITY-SCOPE = \*STD**

The static value of this priority class is \*STD, i.e. the dynamic value of the superordinate node is taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static value of the ROOT node of an LLM is \*STD. The dynamic value in this case is \*NONE (see below).

**HIGH-PRIORITY-SCOPE = \*NONE**

The HIGH-PRIORITY-SCOPE priority class is undefined, i.e. there are no modules which are to be searched before all others for resolving external references. The value of the superordinate node is not taken over.

**HIGH-PRIORITY-SCOPE = <c-string 1..255 with-low> / <text 1..255>**

Path name of the sub-LLM which is to be searched first for resolving external references.

**LOW-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is only to be searched for resolving external references after the search was unsuccessful in all other modules (see [section "Rules for resolving external references" on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

**FORBIDDEN-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is not to be searched for resolving external references (see [section "Rules for resolving external references" on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

An example of the inheritance mechanism for the PRIORITY-SCOPE values is provided in the description of the INCLUDE-MODULES statement on page [221](#).

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled. A name conflict can occur within this statement only if RUN-TIME-VISIBILITY=YES is specified.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## MODIFY-STD-DEFAULTS

This statement modifies the global default values for a BINDER run.

### MODIFY-STD-DEFAULTS

```

OVERWRITE = *UNCHANGED / *YES / *NO
, FOR-BS2000-VERSIONS = *UNCHANGED / *FROM-CURRENT(...) / *FROM-V10(...) /
    *FROM-OSD-V1(...) / *FROM-OSD-V3(...) / *FROM-OSD-V4(...)
    *FROM-CURRENT(...)
    | CONNECTION-MODE = *OSD-DEFAULT / *BY-RELOCATION / *BY-RESOLUTION
    *FROM-V10(...)
    | CONNECTION-MODE = *BY-RESOLUTION / *BY-RELOCATION / *BY-RESOLUTION
    *FROM-OSD-V1(...)
    | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION
    *FROM-OSD-V3(...)
    | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION
    *FROM-OSD-V4(...)
    | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION
, REQUIRED-COMPRESSION = *UNCHANGED / *NO / *YES

```

continued ➔

```

,NAME-COLLISION = *UNCHANGED / *PARAMETERS(...)
  *PARAMETERS(...)
    | INCLUSION = *UNCHANGED / *IGNORED / *WARNING(...) / *ERROR(...)
    | *WARNING(...)
    | | SCOPE = *WHOLE-LLM / *SLICE
    | *ERROR(...)
    | | SCOPE = *WHOLE-LLM / *SLICE
    | ,SAVE = *UNCHANGED / *IGNORED / *WARNING(...) / *ERROR(...)
    | *WARNING(...)
    | | SCOPE = *WHOLE-LLM / *SLICE
    | *ERROR(...)
    | | SCOPE = *WHOLE-LLM / *SLICE
    | ,SYMBOL-PROCESSING = *UNCHANGED / *IGNORED / *WARNING(...) / *ERROR(...)
    | *WARNING(...)
    | | SCOPE = *WHOLE-LLM / *SLICE
    | *ERROR(...)
    | | SCOPE = *WHOLE-LLM / *SLICE

```

**OVERWRITE =**

Specifies whether or not overwriting is permitted.

**OVERWRITE = \*UNCHANGED**

BINDER uses the value of the operand from the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*YES.

**OVERWRITE = \*YES**

Overwriting is permitted.

**OVERWRITE = \*NO**

Overwriting is not permitted.



**FOR-BS2000-VERSIONS =**

Specifies the BS2000 OSD/BC version for which the generated LLM is to be loaded by DBL. DBL can only process the LLM in the specified (or a higher) version of BS2000 OSD/BC.

**FOR-BS2000-VERSIONS = \*BY-PROGRAM**

BINDER defines the format of the generated LLM on the basis of its contents. The LLM format is always the lowest possible that can provide the required functionality. For example, an LLM containing RISC code has format 3, while an LLM containing compressed text is created in format 2.

**FOR-BS2000-VERSIONS = \*UNCHANGED**

BINDER uses the value of the operand from the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-CURRENT**

The BS2000 OSD/BC version under which BINDER is currently running is used.

**CONNECTION-MODE =**

Defines the type of connection between the private and public slices. This operand is only meaningful if the LLM is divided into slices according to the public attribute of the CSECTs it contains.

**CONNECTION-MODE = \*OSD-DEFAULT**

This specification is supported for compatibility reasons. It is equivalent to CONNECTION-MODE = \*BY-RELOCATION.

**CONNECTION-MODE = \*BY-RELOCATION**

The connection between private and public slices is by relocation.

**CONNECTION-MODE = \*BY-RESOLUTION**

The connection between private and public slices is by resolution.

**FOR-BS2000-VERSIONS = \*FROM-V10(...)**

The LLM can be loaded by DBL in any version of BS2000 OSD/BC.

**CONNECTION-MODE = \*BY-RESOLUTION / \*BY-RELOCATION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V1(...)**

The LLM can be loaded by DBL in any version of BS2000 OSD/BC.

**CONNECTION-MODE = \*BY-RESOLUTION / \*BY-RELOCATION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V3(...)**

The LLM can be loaded by DBL in BS2000/OSD-BC V3.0 or higher.

**CONNECTION-MODE = \*BY-RELOCATION / \*BY-RESOLUTION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V4(...)**

The LLM can be loaded by BLSSERV in BS2000/OSD-BC V4.0 or higher.

**CONNECTION-MODE = \*BY-RELOCATION / \*BY-RESOLUTION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**REQUIRED-COMPRESSION =**

Specifies whether the data is to be compressed for better utilization of the disk capacity.

**REQUIRED-COMPRESSION = \*UNCHANGED**

BINDER uses the value specified in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*NO.

**REQUIRED-COMPRESSION = \*NO**

No data compression is carried out.

**REQUIRED-COMPRESSION = \*YES**

Data compression is carried out.

**NAME-COLLISION =**

Specifies how name conflicts are to be handled.

**NAME-COLLISION = \*UNCHANGED**

BINDER uses the value of the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER does not handle name conflicts.

**NAME-COLLISION = \*PARAMETERS(...)****INCLUSION =**

Specifies how name conflicts which occur during inclusion of modules are to be handled.

**INCLUSION = \*UNCHANGED**

BINDER uses the value of the same operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*IGNORED.

**INCLUSION = \*IGNORED**

Specifies that name conflicts which occur during inclusion of modules are to be ignored.

**INCLUSION = \*WARNING(...)**

The user receives a warning if name conflicts occur during inclusion of modules.

**SCOPE =**

Specifies the scope of the definitions for the handling of name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. BINDER ignores name conflicts between different slices.

This value may be specified only for user-defined slices.

**INCLUSION = \*ERROR(...)**

Specifies that the inclusion of modules is to be aborted if name conflicts (correctable errors) occur.

**SCOPE = \*WHOLE-LLM / \*SLICE**

Defines the scope of validity of the definitions for handling name conflicts. See WARNING for the meanings of the operands.

**SAVE =**

Specifies how name conflicts which occur when saving an LMM are to be handled.

**SAVE = \*UNCHANGED**

BINDER uses the value of the same operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, the LLM is not checked for name conflicts.

**SAVE = \*IGNORED**

Specifies that no checks for name conflicts are to be executed when the LLM is stored.

**SAVE = \*WARNING(...)**

The user receives a warning if name conflicts are detected when the LLM is stored.

**SCOPE = \*WHOLE-LLM / \*SLICE**

Defines the scope of validity of the definitions for handling name conflicts. See INCLUSION=WARNING(...) for the meanings of the operands.

**SAVE = ERROR(...)**

Specifies that the storing of an LLM is to be aborted if name conflicts (correctable errors) occur.

**SCOPE = \*WHOLE-LLM / \*SLICE**

Defines the scope of validity of the definitions for handling name conflicts. See INCLUSION=WARNING(...) for the meanings of the operands.

**SYMBOL-PROCESSING =**

Specifies how name conflicts which occur during the handling of symbols are to be handled.

**SYMBOL-PROCESSING = \*UNCHANGED**

BINDER uses the value of the same operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*IGNORED.

**SYMBOL-PROCESSING = \*IGNORED**

Specifies that the statement for symbol handling is to be executed without checking for name conflicts.

**SYMBOL-PROCESSING = \*WARNING(...)**

The user receives a warning if name conflicts are detected during symbol handling.

**SCOPE = \*WHOLE-LLM / \*SLICE**

Defines the scope of validity of the definitions for handling name conflicts. See INCLUSION=WARNING(...) for the meanings of the operands.

**SYMBOL-PROCESSING = \*ERROR(...)**

Specifies that the statement for symbol handling is to be aborted if name conflicts (correctable errors) occur during symbol handling.

**SCOPE = \*WHOLE-LLM / \*SLICE**

Defines the scope of validity of the definitions for handling name conflicts. See INCLUSION=WARNING(...) for the meanings of the operands.

## MODIFY-SYMBOL-ATTRIBUTES

This statement modifies the attributes of control sections (CSECTs) and COMMONs in the current LLM.

The following attributes can be modified (see the “ASSEMBH” manual [2]):

- main memory resident (RESIDENT)
- shareable (PUBLIC)
- read access (READ-ONLY)
- alignment (ALIGNMENT)
- addressing mode (AMODE)
- residence mode (RMODE).

### MODIFY-SYMBOL-ATTRIBUTES

```

SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
,SYMBOL-TYPE = *DEFINITIONS / list-poss(2): *CSECT / *COMMON
,SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM
    *EXPLICIT(...)
        WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> /
            <text 1..255>
        ,EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
,RESIDENT = *UNCHANGED / *YES / *NO
,PUBLIC = *UNCHANGED / *YES / *NO
,READ-ONLY = UNCHANGED / *YES / *NO
,ALIGNMENT = *UNCHANGED / *DOUBLE-WORD / *PAGE / *BUNDLE / <integer 3..12>
,ADDRESSING-MODE = *UNCHANGED / *24 / *31 / *ANY
,RESIDENCY-MODE = *UNCHANGED / *24 / *ANY

```

### SYMBOL-NAME =

Specifies the names of the CSECTs and COMMONs whose attributes are to be modified.

### SYMBOL-NAME = **\*ALL**

The attributes of all CSECTs and COMMONs are modified.

### SYMBOL-NAME = <c-string 1..255>

The names, specified with wildcards, of symbols whose attributes are modified.

Note: BINDER checks special data type <symbol-with-wild> (see [page 199](#)).

**SYMBOL-NAME = <text 1..32>**

Explicit specification of the names of symbols whose attributes are modified.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**SYMBOL-TYPE = \*DEFINITIONS / list-poss(2): \*CSECT / \*COMMON**

Specifies whether only attributes of CSECTs, only attributes of COMMONs, or attributes of both, are modified.

**SCOPE =**

Sets one or more pointers. These point to the sub-LLMs in the logical structure of the LLM, in which attributes are modified.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*WHOLE-LLM is specified, all sub-LLMs are referenced.

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Allows the exclusion of individual pointers from the list specified in the WITHIN-SUB-LLM operand. The path names of the sub-LLMs that are to be ignored should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*NONE is specified, no pointers are excluded.

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are involved in the modification of attributes.

**RESIDENT =**

Specifies whether the RESIDENT attribute is modified.

**RESIDENT = \*UNCHANGED**

The previous value for the RESIDENT attribute is retained.

**RESIDENT = \*YES**

The specified symbols are resident in main memory (class 3 memory).

**RESIDENT = \*NO**

The specified symbols are pageable.

**PUBLIC =**

Specifies whether the PUBLIC attribute is modified.

**PUBLIC = \*UNCHANGED**

The previous value for the PUBLIC attribute is retained.

**PUBLIC = \*YES**

The specified symbols are shareable.

**PUBLIC = \*NO**

The specified symbols are not shareable.

**READ-ONLY =**

Specifies whether the READ-ONLY attribute is modified.

**READ-ONLY = \*UNCHANGED**

The previous value for the READ-ONLY attribute is retained.

**READ-ONLY = \*YES**

Only read access is permitted for the specified symbols. The entry YES is not permitted for a vector program.

**READ-ONLY = \*NO**

Read and write access are permitted for the specified symbols.

**ALIGNMENT =**

Specifies whether or not the alignment is modified.

**ALIGNMENT = \*UNCHANGED**

The previous value for the alignment is retained.

**ALIGNMENT = \*DOUBLE-WORD**

The specified symbols are aligned on a doubleword boundary.

**ALIGNMENT = \*PAGE**

The specified symbols are aligned on a page boundary, i.e. the address is a multiple of 4096 (X'1000').

**ALIGNMENT = \*BUNDLE**

The new CSECT is aligned on an address that is a multiple of 16.

**ALIGNMENT = <integer 3..12>**

The specified symbols are aligned on an address that is a multiple of  $2^n$ . The exponent "n" is specified by <integer 3..12>.

**ADDRESSING-MODE =**

Specifies whether the addressing mode (AMODE) is modified. If an inconsistency occurs between the original and specified values for AMODE and RMODE, the operand will be ignored.

**ADDRESSING-MODE = \*UNCHANGED**

The previous value for the addressing mode is retained.

**ADDRESSING-MODE = \*24**

The 24-bit addressing mode is assigned to the specified symbols.

**ADDRESSING-MODE = \*31**

The 31-bit addressing mode is assigned to the specified symbols.

**ADDRESSING-MODE = \*ANY**

The 24- or 31-bit addressing mode is assigned to the specified symbols. The decision concerning the addressing mode is not made until load time.

**RESIDENCY-MODE =**

Specifies whether or not the residence mode (RMODE) is modified. If an inconsistency occurs between the original and specified values for AMODE and RMODE, the operand will be ignored.

**RESIDENCY-MODE = \*UNCHANGED**

The previous value for the residence mode is retained.

**RESIDENCY-MODE = \*24**

The specified symbols may only be loaded below 16 Mb.

**RESIDENCY-MODE = \*ANY**

The specified symbols may be loaded below and above 16 Mb.



## MODIFY-SYMBOL-TYPE

This statement defines the types of symbols, where “symbols” means only external references (EXTRNs), weak external references (WXTRNs) and V constants (VCONs). The following conversions are possible:

EXTRN           → WXTRN or VCON  
 WXTRN           → EXTRN or VCON  
 VCON             → EXTRN or WXTRN.

EXTRN and WXTRN, which are not referenced, cannot be converted to VCON. The scope of validity of the definitions can be restricted with the operand SCOPE. This statement has no effect on previously resolved external references.

### MODIFY-SYMBOL-TYPE

```

SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
, SYMBOL-TYPE = *REFERENCES / list-poss(3): *EXTRN / *VCON / *WXTRN
, SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM
  *EXPLICIT(...)
    |
    | WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> /
    |                   <text 1..255>
    |
    | ,EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
    |
,NEW-SYMBOL-TYPE = *EXTRN / *VCON / *WXTRN

```

### **SYMBOL-NAME =**

Specifies the symbols whose types are to be modified.

### **SYMBOL-NAME = \*ALL**

All symbols in the scope of validity defined with SCOPE are to be processed.

### **SYMBOL-NAME = list-poss(40): <c-string 1..255> / <text 1..32>**

Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

### **SYMBOL-TYPE = \*REFERENCES / list-poss(3): \*EXTRN / \*VCON / \*WXTRN**

Specifies which types of external references are to be processed. If REFERENCES is specified, all external references are processed.

**SCOPE =**

Defines the scope of validity by setting one or more pointers. These pointers point to the sub-LLMs in the logical structure of the LLM in which the symbol types are to be modified.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM  
(see the BEGIN-SUB-LLM-STATEMENTS statement).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs are specified in <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

\*WHOLE-LLM addresses all sub-LLMs.

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Permits individual pointers to be excluded from the list entered for the WITHIN-SUB-LLM operand. The path names of the sub-LLMs to be excluded are specified in <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

\*NONE specifies that no pointers are to be excluded.

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are affected by the symbol type modification.

**NEW-SYMBOL-TYPE = \*EXTRN / \*VCON / \*WXTRN**

Specifies the new type which is to be assigned to the symbols.

## MODIFY-SYMBOL-VISIBILITY

This statement specifies the extent to which control sections (CSECTs) and entry points (ENTRYS) in the current LLM are to remain visible or be masked for a subsequent BINDER or DBL run. Masked symbols will not be found for resolving external references in subsequent BINDER or DBL runs.

### MODIFY-SYMBOL-VISIBILITY

```

SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
, SYMBOL-TYPE = *DEFINITIONS / list-poss(3): *CSECT / *ENTRY / *COMMON
, SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM
  *EXPLICIT(...)
    |
    | WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> /
    | <text 1..255>
    |
    | , EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
  , VISIBLE = *YES / *NO(...)
    *NO(...)
      |
      | KEEP-RESOLUTION = *YES / *NO
  , NAME-COLLISION = STD / *IGNORED / *WARNING(...) / *ERROR(...)
    *WARNING(...)
      |
      | SCOPE = *WHOLE-LLM / *SLICE
    *ERROR(...)
      |
      | SCOPE = *WHOLE-LLM / *SLICE

```

#### **SYMBOL-NAME =**

Defines the symbols whose masking is to be modified.

#### **SYMBOL-NAME = \*ALL**

The masking of all symbols is modified.

#### **SYMBOL-NAME = list-poss(40): <c-string 1..255>/ <text 1..32>**

Explicit specification of the symbols whose masking is to be modified.

Wildcards may be specified.

Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

#### **SYMBOL-TYPE = \*DEFINITIONS / list-poss(3): \*CSECT / \*ENTRY / \*COMMON**

Defines the type of the symbols whose masking is modified. When \*DEFINITIONS is specified, the masking of all specified symbols is modified.

**SCOPE =**

Sets one or more pointers. These point to the sub-LLMs in the logical structure of the LLM in which the masking of symbols is modified.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*WHOLE-LLM is specified, all sub-LLMs are referenced.

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Allows the exclusion of individual pointers from the list specified in the WITHIN-SUB-LLM operand. The path names of the sub-LLMs that are to be ignored should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*NONE is specified, no pointers are excluded.

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are involved in the modification of masking of symbols.

**VISIBLE =**

Determines whether the specified symbols are to remain visible or be masked.

**VISIBLE = \*YES**

The specified symbols remains visible for subsequent BINDER or DBL runs.

**VISIBLE = \*NO(...)**

The specified symbols are masked for subsequent BINDER or DBL runs.

**KEEP-RESOLUTION =**

Determines whether resolved external references for the specified symbol are to remain resolved or be deresolved.

**KEEP-RESOLUTION = \*YES**

Resolved external references remain resolved.

**KEEP-RESOLUTION = \*NO**

Resolved external references are deresolved.

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## REMOVE-MODULES

This statement removes modules from the current LLM. Object modules and sub-LLMs can be removed. The following are not removed:

- the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement)
- a sub-LLM whose beginning is defined with the BEGIN-SUB-LLM-STATEMENTS statement but whose end has not yet been defined with the END-SUB-LLM-STATEMENTS statement.

The user can use the path name to specify from which sub-LLM of the current LLM the modules are to be removed.

### REMOVE-MODULES

```
NAME = *ALL / list-poss(40): <c-string 1..32 with-low> / <text 1..32>
, PATH-NAME = *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
```

#### **NAME =**

Defines the modules of the current LLM that are to be removed.

#### **NAME = \*ALL**

All modules of the current LLM are removed.

#### **NAME = <structured-name 1..32>**

Explicit specification of the modules that are to be removed.

#### **NAME = <text 1..32>**

A logical name which was specified in an INCLUDE-MODULES or REPLACE-MODULES statement.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

#### **PATH-NAME =**

Defines the sub-LLM in the logical structure of the current LLM in the work area, from which modules are to be removed.

#### **PATH-NAME = \*CURRENT-SUB-LLM**

The current sub-LLM is assumed (see BEGIN-SUB-LLM-STATEMENTS statement).

#### **PATH-NAME = <text 1..255>**

Path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

## RENAME-SYMBOLS

This statement replaces the names of program definitions and references in the current LLM with a new name.

A separate RENAME-SYMBOLS statement must be entered for each symbol name which is to be changed. The user can specify the types of symbols with this name which are to be processed.

The following program definitions can be renamed:

- control sections (CSECTs)
- entry points (ENTRYs)
- COMMONs.

The following references can be renamed:

- external references (EXTRNs)
- V-type constants
- weak external references (WXTRNs).

**RENAME-SYMBOLS**

```

SYMBOL-NAME = <c-string 1..32 with-low> / <text 1..32>
, SYMBOL-TYPE = *ALL / *DEFINITIONS / *REFERENCES / list-poss(6): *CSECT / *ENTRY / *COMMON /
*EXTRN / *VCON / *WXTRN
, SYMBOL-OCCURRENCE = *PARAMETERS (...)
  *PARAMETERS(...)
    | FIRST-OCCURRENCE = 1 / <integer 1..32767>
    | , OCCURRENCE-NUMBER = 1 / <integer 1..32767> / *ALL
    | , WARNING-MESSAGE = *EXCEED / *LOWER / *BOTH
, SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM
  *EXPLICIT(...)
    | WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> /
    | <text 1..255>
    | , EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
, NEW-NAME = <c-string 1..32 with-low> / <text 1..32>
, NAME-COLLISION = *STD / *IGNORED / *WARNING(...) / *ERROR(...)
  *WARNING(...)
    | SCOPE = *WHOLE-LLM / *SLICE
  *ERROR(...)
    | SCOPE = *WHOLE-LLM / *SLICE

```

**SYMBOL-NAME = <text 1..32> / <c-string 1..32>**

Defines the symbols to be renamed.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**SYMBOL-TYPE =**

Defines the type of the symbols that are to be renamed.

**SYMBOL-TYPE = \*ALL**

All types of symbols are renamed.

**SYMBOL-TYPE = \*DEFINITIONS**

Program definitions are renamed.

**SYMBOL-TYPE = \*REFERENCES**

References are renamed.

**SYMBOL-TYPE = \*CSECT**

Control sections are renamed.



**SYMBOL-TYPE = \*ENTRY**

Entry points are renamed.

**SYMBOL-TYPE = \*COMMON**

COMMONs are renamed.

**SYMBOL-TYPE = \*EXTRN**

External references are renamed.

**SYMBOL-TYPE = \*VCON**

V-type constants are renamed.

**SYMBOL-TYPE = \*WXTRN**

Weak external references are renamed.

**SYMBOL-OCCURRENCE = \*PARAMETERS(...)**

Defines the position and number of occurrences of the symbols. In the logical structure, the symbols are searched in the following order:

1. CSECTs and ENTRYs
2. COMMONs
3. EXTRN, VCON and WXTRN.

**FIRST-OCCURRENCE = 1 / <integer 1..32767>**

Defines the position of the first occurrence of the symbol with the specified name. Renaming begins at the xth occurrence of this symbol (x: <integer 1..32767>).

**OCCURRENCE-NUMBER = 1 / <integer 1..32767>**

Defines the number of the occurring symbols which are to be renamed.

If ALL is specified, every occurrence of the symbol after the position specified with FIRST-OCCURRENCE is renamed. Otherwise, the next y (y: <integer 1..32767>) occurrences of the symbols are renamed.

**WARNING-MESSAGE =**

Defines when the warning messages are to be sent.

**WARNING-MESSAGE = \*EXCEED**

The message BND2103 is sent when the number of occurrences, given in the OCCURRENCE-NUMBER operand, exceeds the real number of occurrences found in the given scope.

**WARNING-MESSAGE = \*LOWER**

The message BND2106 is sent when the number of occurrences, given in the OCCURRENCE-NUMBER operand, is lower than the real number of occurrences found in the given scope.

**WARNING-MESSAGE = \*BOTH**

The message BND2103 and the message BND2106 are sent.

**SCOPE =**

Sets one or more pointers. These point to the sub-LLMs within the logical structure of the LLM in which BINDER is to rename symbols.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs should be specified for <text 1..255>.

When \*WHOLE-LLM is specified, all sub-LLMs are referenced.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Allows the exclusion of individual sub-LLMs from the list specified in the WITHIN-SUB-LLM operand. The path names of the sub-LLMs that are to be ignored should be specified for <text 1..255>.

When \*NONE is specified, no pointers are excluded.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are involved in the renaming of symbols.

**NEW-NAME = <text 1..32> / <c-string 1..32>**

The new symbol name which will replace the name specified for SYMBOL-NAME.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## REPLACE-MODULES

This statement replaces one or more modules in the current LLM with new modules. Both object modules and sub-LLMs can replace modules in the current LLM.

Object modules, LLMs or both can be replaced as modules. LLMs with user-defined slices and LLMs without relocation information cannot be replaced. Either whole LLMs can be replaced or sub-LLMs can be selected.

The input source may be:

- for object modules: a program library (element type R), an object module library or the EAM object module file
- for LLMs and sub-LLMs: a program library (element type L).

### REPLACE-MODULES

```

NAME = <c-string 1..32 with-low> / <text 1..32>
, PATH-NAME = *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
, MODULE-CONTAINER = *LIBRARY-ELEMENT (...) / *FILE(...) / *OMF(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY = *CURRENT-INPUT-LIB / <filename 1..54 without-gen-vers> / *LINK(...) /
    | *BLSLIB-LINK / *OMF
    | *LINK(...)
    | | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
    | , ELEMENT = *ALL (...) / list-poss(40): <composed-name 1..64>(…) / <c-string 1..64>(…)
    | *ALL(…)
    | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
    | <composed-name>(…)
    | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
    | | , SUB-LLM = *WHOLE-LLM / <c-string 1..255 with-low> / <text 1..255>
    | <c-string>(…)
    | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
    | | , SUB-LLM = *WHOLE-LLM / <c-string 1..255 with-low> / <text 1..255>
    | , TYPE = (*L *R) / list-poss(2): *L / *R

```

continued ➡

<pre> *FILE(...)         FILE-NAME = &lt;filename 1..54 without-gen&gt; / *LINK(...)         *LINK(...)                 LINK-NAME = &lt;structured-name 1..8&gt; / &lt;filename 1..8 without-gen&gt;                 ,SUB-LLM = *WHOLE-LLM / &lt;c-string 1..255 with-low&gt; / &lt;text 1..255&gt;     *OMF(...)         ELEMENT = *ALL / list-poss(40): &lt;composed-name 1..64&gt; / &lt;c-string 1..64&gt;         ,NEW-NAME = *INTERNAL / *ELEMENT-NAME / &lt;c-string 1..32 with-low&gt; / &lt;text 1..32&gt;         ,SLICE = *CURRENT-SLICE / *ROOT / &lt;structured-name 1..32&gt;         ,LOGICAL-STRUCTURE = INCLUSION-DEFAULT / *WHOLE-LLM / *OBJECT-MODULES         ,TEST-SUPPORT = *INCLUSION-DEFAULT / *NO / *YES         ,RUN-TIME-VISIBILITY = UNCHANGED / *NO / *YES         ,RESOLUTION-SCOPE = *UNCHANGED / *STD / *PARAMETERS(...)         *PARAMETERS(...)                 HIGH-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / &lt;text 1..255&gt;                 ,LOW-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / &lt;text 1..255&gt;                 ,FORBIDDEN-SCOPE = *UNCHANGED / *STD / *NONE / &lt;text 1..255&gt;         ,NAME-COLLISION = *STD / *IGNORED / *WARNING(...) / *ERROR(...)         *WARNING(...)                 SCOPE = WHOLE-LLM / *SLICE         *ERROR(...)                 SCOPE = WHOLE-LLM / *SLICE </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**NAME =**

Specifies the name of the module that is to be replaced in the current LLM.

**NAME = <structured-name 1..32>**

Explicit specification of the name of the module to be replaced.

**NAME = <text 1..32>**

The logical name assigned to the module in an INCLUDE-MODULES statement or in a previous REPLACE-MODULES statement (NEW-NAME operand).

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**PATH-NAME =**

Defines the sub-LLM in the logical structure of the current LLM in which modules are replaced.

**PATH-NAME = \*CURRENT-SUB-LLM**

The current sub-LLM is assumed (see BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = <text 1..255>**

Path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**MODULE-CONTAINER =**

Defines where the LLM is stored.

**MODULE-CONTAINER = \*LIBRARY-ELEMENT(...)**

The LLM is stored in a program library.

**LIBRARY =**

Specifies the input source from which the modules are read.

**LIBRARY = \*CURRENT-INPUT-LIB**

The input source from which the *last* module (OM or LLM) was read by means of a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES statement is used. The scope of the operand relates to one edit run.

**LIBRARY = <filename 1..54 without-gen-vers>**

File name of the library that is to be used as the input source.

**LIBRARY = \*LINK(...)**

Denotes a library by means of the file link name.

**LINKNAME = <structured-name 1..8>**

File link name of the library that is to be used as the input source.

**LIBRARY = \*BLSLIB-LINK**

The input sources are libraries with the file link name BLSLIBnn (00≤nn≤99). The libraries are searched in *ascending* order of “nn” values for the file link name.

**LIBRARY = \*OMF**

The input source is the EAM object module file. This contains only object modules.

**ELEMENT =**

Defines the element name and the element version of the modules that are read from the specified input source.

**ELEMENT = \*ALL(...)**

All modules are read from the specified input source.

**VERSION =**

Specifies the element version of the module. The element version is applicable only to program libraries.

**VERSION = \*HIGHEST-EXISTING**

BINDER takes as element version the default value for the highest version in the case of program libraries (see the “LMS” manual [3]).

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**ELEMENT = <composed-name 1..64>(...)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)).

**VERSION = \*HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>**

Specifies the element version of the module. The element version is applicable only to program libraries.

See above for meaning of operands.

**SUB-LLM =**

Specifies whether the whole LLM or a sub-LLM is taken over as an element.

**SUB-LLM = \*WHOLE-LLM**

The whole LLM is taken over.

**SUB-LLM = <text 1..255>**

Path name of the sub-LLM that is taken over.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**ELEMENT = <c-string 1..64>(...)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)).

See above for meaning of operands.

**TYPE =**

Defines the priority of the modules (object modules and/or LLMs) that can replace a module in the current LLM.

**TYPE = (\*L,\*R)**

Both LLMs and object modules can replace modules in the current LLM. If the same name is specified for an LLM as for an object module, the LLM is taken over.

**TYPE = (\*R,\*L)**

Both LLMs and object modules can replace modules. If the same name is specified for an LLM as for an object module, the object module is taken over.

**TYPE = \*R**

Only object modules can replace modules.

**TYPE = \*L**

Only LLMs can replace modules.

**MODULE-CONTAINER = \*FILE(...)****FILE-NAME =**

Specifies the PAM file that contains the LLM.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Name of the PAM file in which the LLM is stored.

**FILE-NAME = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the PAM file in which the LLM is stored.

**MODULE-CONTAINER = \*OMF(...)**

The input source is the EAM object module file. This contains only object modules.

**ELEMENT = \*ALL / list-poss(40): <composed-name 1..64> / <c-string 1..64>**

See above for the meanings of the operands.

**NEW-NAME =**

Specifies the new logical name for the replaced module.

**NEW-NAME = \*INTERNAL**

The internal name is used as the logical name.

**NEW-NAME = \*ELEMENT-NAME**

The name of the library element is used as the new logical name.

**NEW-NAME = <structured-name 1..32>**

Explicit specification of the new logical name.



**SLICE =**

Specifies the slice in the physical structure of the LLM in which the modules are replaced. The slice must be defined with a SET-USER-SLICE-POSITION statement.

**SLICE = \*CURRENT-SLICE**

Modules are replaced in the current slice. This is the slice that was defined by the most recent preceding SET-USER-SLICE-POSITION statement.

**SLICE = \*ROOT**

Modules are replaced in the root slice (%ROOT).

**SLICE = <structured-name 1..32>**

Explicit specification of the slice in which modules are replaced.

**LOGICAL-STRUCTURE =**

Specifies whether the logical structure information from the modules is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

The logical structure information is not taken over. A structure comprising only object modules (OMs) is set up in the current LLM.

**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is taken over into the current LLM.



Information on the existence of the list for symbolic debugging is provided in the "T&D" column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.

**RUN-TIME-VISIBILITY =**

Specifies whether the module is to be regarded as a runtime module. All symbols in a runtime module are masked when the module is stored and are, for the moment, not used for resolving external references. This masking of the symbols is canceled during any subsequent read access to the module (e.g. with START-LLM-UPDATE or INCLUDE-MODULES).

**RUN-TIME-VISIBILITY = \*UNCHANGED**

The value is not changed. When a module is included in an LLM for the first time with INCLUDE-MODULES or REPLACE-MODULES, BINDER assumes the value NO.

**RUN-TIME-VISIBILITY = \*NO**

The module is not to be regarded as a runtime module.

**RUN-TIME-VISIBILITY = \*YES**

The module is to be regarded as a runtime module. All symbols in the module are masked when the module is stored.

**RESOLUTION-SCOPE =**

Defines priority classes that control the order in which BINDER is to search other modules when resolving external references. Two values must be discriminated for each class:

- The dynamic value influences the order in which modules are searched to resolve external references. It is, however, not stored in the LLM.
- The static value is stored in the LLM and forms the basis for determining the dynamic value.

**RESOLUTION-SCOPE = \*UNCHANGED**

The static values of the priority classes are those stored in the modules concerned. They are assigned the value \*STD (see below) for object modules (OMs).

**RESOLUTION-SCOPE = \*STD**

The static values of the priority classes are \*STD, i.e. the dynamic values of superordinate nodes are taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static values of the ROOT node of an LLM is \*STD. The dynamic values in this case are \*NONE (see below).

**RESOLUTION-SCOPE = \*PARAMETERS(...)**

The static values of the individual priority classes are defined separately.

**HIGH-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is to be searched before all others for resolving external references (see [section “Rules for resolving external references” on page 81](#)).

**HIGH-PRIORITY-SCOPE = \*STD**

The static value of this priority class is \*STD, i.e. the dynamic value of the superordinate node is taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static value of the ROOT node of an LLM is \*STD. The dynamic value in this case is \*NONE (see below).

**HIGH-PRIORITY-SCOPE = \*NONE**

The HIGH-PRIORITY-SCOPE priority class is undefined, i.e. there are no modules which are to be searched before all others for resolving external references. The value of the superordinate node is not taken over.

**HIGH-PRIORITY-SCOPE = <c-string 1..255 with-low> / <text 1..255>**

Path name of the sub-LLM which is to be searched first for resolving external references.

**LOW-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is only to be searched for resolving external references after the search was unsuccessful in all other modules (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

**FORBIDDEN-SCOPE = \*UNCHANGED / \*STD / \*NONE /  
<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is not to be searched for resolving external references (see [section “Rules for resolving external references” on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

An example of the inheritance mechanism for the PRIORITY-SCOPE values is provided in the description of the RESOLVE-BY-AUTOLINK statement on page [283](#).

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value \*IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## RESOLVE-BY-AUTOLINK

This statement determines how BINDER is to resolve unresolved external references (autolink function).

BINDER searches the specified libraries for modules with suitable CSECTs and ENTRYs and includes the located modules in the current LLM.

LLMs with user-defined slices and LLMs without relocation information are not included. If BINDER finds such LLMs, it aborts the autolink function.

### RESOLVE-BY-AUTOLINK

```

LIBRARY = *CURRENT-INPUT-LIB / *BLSLIB-LINK /
           list-poss(40): <filename 1..54 without-gen-vers> / *LINK(...)

*LINK(...)
  | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
, TYPE = (*L,*R) / list-poss(2): *L / *R
, SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
, SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM

*EXPLICIT(...)
  | WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
  | EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
, NAME = *INTERNAL / *ELEMENT-NAME
, PATH-NAME = *CURRENT-SUB-LLM / <c-string 1..255 with-low> / <text 1..255>
, LOGICAL-STRUCTURE = *INCLUSION-DEFAULT / *WHOLE-LLM / *OBJECT-MODULES
, TEST-SUPPORT = *INCLUSION-DEFAULT / *NO / *YES
, RUN-TIME-VISIBILITY = *UNCHANGED / *NO / *YES
, RESOLUTION-SCOPE = *UNCHANGED / *STD / *PARAMETERS(...)

*PARAMETERS(...)
  | HIGH-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>
  | LOW-PRIORITY-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>
  | FORBIDDEN-SCOPE = *UNCHANGED / *STD / *NONE / <text 1..255>

```

continued ➔

```
,NAME-COLLISION = STD / *IGNORED / *WARNING(...) / *ERROR(...)
    *WARNING(...)
    |   SCOPE = *WHOLE-LLM / *SLICE
    *ERROR(...)
    |   SCOPE = *WHOLE-LLM / *SLICE
```

**LIBRARY =**

Specifies one or more input libraries that BINDER is to search for resolving external references.

**LIBRARY = \*CURRENT-INPUT-LIB**

The input library from which the *last* module (OM or LLM) was read by means of a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES statement is searched. The scope of the operand relates to one edit run.

**LIBRARY = \*BLSLIB-LINK**

The input libraries with the file link names BLSLIBnn (00≤nn≤99) are searched.

**LIBRARY = <filename 1..54 without-gen-vers>**

File names of the input libraries that are searched.

**LIBRARY = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the input library that is searched.

**TYPE =**

Defines the priority of the modules (object modules and/or LLMs) that are searched for suitable CSECTs and ENTRYs.

**TYPE = (\*L,\*R)**

Both LLMs and object modules are searched. If the same name is present for an LLM and for an object module, the LLM is searched.

**TYPE = (\*R,\*L)**

Both LLMs and object modules are searched. If the same name is present for an LLM and for an object module, the object module is searched.

**TYPE = \*R**

Only object modules are searched.

**TYPE = \*L**

Only LLMs are searched.

**SYMBOL-NAME =**

Defines the external references that BINDER is to resolve.

**SYMBOL-NAME = \*ALL**

All external references are to be resolved.

**SYMBOL-NAME = <c-string 1..255> / <text 1..32>**

Names of the external references that are to be resolved. Wildcards may be specified.

Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

**SCOPE =**

Sets one or more pointers. These point to the sub-LLMs in the logical structure of the LLM, in which BINDER is to resolve unresolved external references.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement on [page 209](#)).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

If \*WHOLE-LLM is specified, all sub-LLMs are referenced.

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Allows the exclusion of individual pointers from the list specified in the WITHIN-SUB-LLM operand. The path names of the sub-LLMs that are to be ignored should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

If \*NONE is specified, no pointers are excluded.

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are involved in the resolution of external references.

**NAME =**

Specifies the name to be used as the logical name when the module is included by the autolink function.

**NAME = \*INTERNAL**

The internal name is used as the logical name.

**NAME = \*ELEMENT-NAME**

The name of the library element containing the module is used as the logical name. If necessary, BINDER truncates this name to 32 characters.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**PATH-NAME =**

Defines the sub-LLM in the logical structure of the current LLM in the work area in which modules are included.

**PATH-NAME = \*CURRENT-SUB-LLM**

The *current sub-LLM* is assumed (see BEGIN-SUB-LLM-STATEMENTS statement).

**PATH-NAME = <text 1..255>**

Path name of the sub-LLM in the logical structure of the current LLM.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**LOGICAL-STRUCTURE =**

Specifies whether the logical structure information from the modules is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**LOGICAL-STRUCTURE = WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = OBJECT-MODULES**

The logical structure information is not taken over. A structure comprising only object modules (OMs) is established in the current LLM.



**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is taken over into the current LLM.



Information on the existence of the list for symbolic debugging is provided in the “T&D” column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*INCLUSION-DEFAULT**

The values of the INCLUSION-DEFAULTS operand from the START-LLM-CREATION, START-LLM-UPDATE or MODIFY-LLM-ATTRIBUTES statements from the same edit run are assumed.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.

**RUN-TIME-VISIBILITY =**

Specifies whether the symbols in a runtime module are masked when the module is stored and are, for the moment, not used for resolving external references. This masking of the symbols is canceled during any subsequent read access to the module (e.g. with START-LLM-UPDATE or INCLUDE-MODULES).

**RUN-TIME-VISIBILITY = \*UNCHANGED**

The value is not changed. When a module is included in an LLM for the first time with INCLUDE-MODULES or REPLACE-MODULES, BINDER assumes the value NO.

**RUN-TIME-VISIBILITY = \*NO**

Symbols that are resolved by autolink remain visible.

**RUN-TIME-VISIBILITY = \*YES**

Symbols that are resolved by autolink module are masked when the module is stored.

**RESOLUTION-SCOPE =**

Defines priority classes that control the order in which BINDER is to search other modules when resolving external references. Two values must be discriminated for each class:

- The dynamic value  
influences the order in which modules are searched to resolve external references. It is, however, not stored in the LLM.
- The static value  
is stored in the LLM and forms the basis for determining the dynamic value.

**RESOLUTION-SCOPE = \*UNCHANGED**

The static values of the priority classes are those stored in the modules concerned. They are assigned the value \*STD (see below) for object modules (OMs).

**RESOLUTION-SCOPE = \*STD**

The static values of the priority classes are \*STD, i.e. the dynamic values of superordinate nodes are taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static values of the ROOT node of an LLM is \*STD. The dynamic values in this case are \*NONE (see below).

**RESOLUTION-SCOPE = \*PARAMETERS(...)**

The static values of the individual priority classes are defined separately.

**HIGH-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is to be searched before all others for resolving external references (see [section "Rules for resolving external references" on page 81](#)).

**HIGH-PRIORITY-SCOPE = \*STD**

The static value of this priority class is \*STD, i.e. the dynamic value of the superordinate node is taken over into the logical LLM structure. This inheritance mechanism is applied each time a new search is started for resolving external references.

The default value for the static value of the ROOT node of an LLM is \*STD. The dynamic value in this case is \*NONE (see below).

**HIGH-PRIORITY-SCOPE = \*NONE**

The HIGH-PRIORITY-SCOPE priority class is undefined, i.e. there are no modules which are to be searched before all others for resolving external references. The value of the superordinate node is not taken over.

**HIGH-PRIORITY-SCOPE = <c-string 1..255 with-low> / <text 1..255>**

Path name of the sub-LLM which is to be searched first for resolving external references.

**LOW-PRIORITY-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is only to be searched for resolving external references after the search was unsuccessful in all other modules (see [section "Rules for resolving external references" on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

**FORBIDDEN-SCOPE = \*UNCHANGED / \*STD / \*NONE /**

**<c-string 1..255 with-low> / <text 1..255>**

Defines which sub-LLM is not to be searched for resolving external references (see [section "Rules for resolving external references" on page 81](#)).

The meanings of the separate operand values are analogous to the priority class HIGH-PRIORITY-SCOPE.

The following example illustrates the inheritance mechanism for the PRIORITY-SCOPE values:

LLM1 and LLM2 are both in a library. Static HIGH-PRIORITY-SCOPE values are defined for both LLMs, as shown in the following table.

<b>Module</b>	<b>Dynamic value of HIGH-PRIORITY-SCOPE</b>
LLM1	LLM1.OM12
└─ OM11	*STD
└─ OM12	*NONE
LLM2	*STD
└─ OM21	LLM2.OM22
└─ OM22	*STD

LLM2 is linked into LLM1. During linkage, BINDER uses the static HIGH-PRIORITY-SCOPE values to determine the dynamic values which influence the search order when resolving external references.

The following table shows the resulting module structure and dynamic values of HIGH-PRIORITY-SCOPE.

<b>Module</b>	<b>Dynamic value of HIGH-PRIORITY-SCOPE</b>	<b>Comment</b>
LLM1	LLM1.OM12	= static value
└─ OM11	*STD	inherited from LLM1
└─ OM12	*NONE	= static value
└─ LLM2	*STD	inherited from LLM1
└─ OM21	LLM2.OM22	= static value
└─ OM22	*STD	inherited from LLM1

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the statement are to be handled.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

## SAVE-LLM

This statement saves the current LLM, created with a START-LLM-CREATION statement or updated with a START-LLM-UPDATE statement, as a type L element in a program library.

An LLM updated by means of the START-LLM-UPDATE statement is written back to the original element during saving if the new element retains the same element name and the same element version in the program library and OVERWRITE=\*YES is specified.

The SAVE-LLM statement does not terminate the BINDER run. It is therefore possible to further process the current LLM in the same BINDER run, or to create a new LLM with another START-LLM-CREATION statement, or to modify a new LLM with another START-LLM-UPDATE statement.

### SAVE-LLM

**MODULE-CONTAINER** = \*CURRENT / \*LIBRARY-ELEMENT(...) / \*FILE(...)

\*LIBRARY-ELEMENT(...)

**LIBRARY** = \*CURRENT / <filename 1..54 without-gen-vers> / \*LINK(...)

\*LINK(...)

| **LINK-NAME** = <structured-name 1..8> / <filename 1..8 without-gen>

,**ELEMENT** = \*CURRENT-NAME (...) / \*INTERNAL-NAME(...) / <composed-name 1..64>(…) /  
<c-string 1..64>(…)

\*CURRENT-NAME(...)

| **VERSION** = \*CURRENT-VERSION / \*INTERNAL-VERSION / \*UPPER-LIMIT /  
\*INCREMENT / <composed-name 1..24> / <c-string 1..24>

\*INTERNAL-NAME(...)

| **VERSION** = \*INTERNAL-VERSION / \*UPPER-LIMIT / \*INCREMENT /  
<composed-name 1..24> / <c-string 1..24>

<composed-name 1..64>(…)

| **VERSION** = \*INTERNAL-VERSION / \*UPPER-LIMIT / \*INCREMENT /  
<composed-name 1..24> / <c-string 1..24>

<c-string 1..64>(…)

| **VERSION** = \*INTERNAL-VERSION / \*UPPER-LIMIT / \*INCREMENT /  
<composed-name 1..24> / <c-string 1..24>

continued ➡

```

*FILE(...)
  |
  | FILE-NAME = *CURRENT / <filename 1..54 without-gen> / *LINK(...)
  |
  | *LINK(...)
  |   |
  |   | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
  |
, OVERWRITE = *LAST-SAVE / *STD / *YES / *NO
, FOR-BS2000-VERSIONS = *LAST-SAVE / *STD / *FROM-CURRENT(...) / *FROM-V10(...) /
  *FROM-OSD-V1(...) / *FROM-OSD-V3(...) / *FROM-OSD-V4(...)

*FROM-CURRENT(...)
  |
  | CONNECTION-MODE = *OSD-DEFAULT / *BY-RELOCATION / *BY-RESOLUTION

*FROM-V10(...)
  |
  | CONNECTION-MODE = *BY-RESOLUTION / *BY-RELOCATION

*FROM-OSD-V1(...)
  |
  | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION

*FROM-OSD-V3(...)
  |
  | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION

*FROM-OSD-V4(...)
  |
  | CONNECTION-MODE = *BY-RELOCATION / *BY-RESOLUTION

, REQUIRED-COMPRESSION = *LAST-SAVE / *STD / *NO / *YES
, NAME-COLLISION = *LAST-SAVE / *STD / *IGNORED / *WARNING(...) / *ERROR(...)

*WARNING(...)
  |
  | SCOPE = *WHOLE-LLM / *SLICE

*ERROR(...)
  |
  | SCOPE = WHOLE-LLM / *SLICE

, SYMBOL-DICTIONARY = *LAST-SAVE / *YES / *NO
, RELOCATION-DATA = *LAST-SAVE / *YES / *NO / *UNRESOLVED-ONLY
, LOGICAL-STRUCTURE = *LAST-SAVE / *WHOLE-LLM / *OBJECT-MODULES / *NONE
, TEST-SUPPORT = *LAST-SAVE / *YES / *NO

```

continued ➔

```

,LOAD-ADDRESS = *LAST-SAVE / *UNDEFINED / *NULL / *BY-SLICES(...) / <x-string 1..8>
  *BY-SLICES(...)
    ADDRESSES = list-poss(40): *REGION(...) / *SLICE(...)
      *REGION(...)
        REGION-NAME = <structured-name 1..32>
        ,REGION-ADDRESS = <x-string 1..8>
      *SLICE(...)
        SLICE-NAME = *ROOT / <structured-name 1..32>
        ,SLICE-ADDRESS = <x-string 1..8> / *BEHIND-SLICE(...)
          *BEHIND-SLICE(...)
            SLICE = <structured-name 1..32>
,ENTRY-POINT = *LAST-SAVE / *STD / *BY-MODULE(...) / <c-string 1..32 with-low> / <text 1..32>
  *BY-MODULE(...)
    PATH-NAME = <text 1..255>
,MAP = *LAST-SAVE / *YES / *NO

```

**MODULE-CONTAINER =**

Defines where the LLM is to be stored.

**MODULE-CONTAINER = \*CURRENT**

The LLM is stored in the library or PAM file specified in the most recent SAVE-LLM or START-LLM-UPDATE statement.

**MODULE-CONTAINER = \*LIBRARY-ELEMENT(...)**

The LLM is to be stored in a program library.

**LIBRARY =**

Defines the program library in which the LLM is saved.

**LIBRARY = \*CURRENT**

BINDER takes the program library from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER selects the program library as follows:

- it takes the library from the associated START-LLM-UPDATE statement if the LLM was updated by means of a START-LLM-UPDATE statement
- it outputs an error message and requests the explicit specification of the library if the LLM was created by means of a START-LLM-CREATION statement.

**LIBRARY = <filename 1..54 without-gen-vers>**

File name of the program library in which the LLM is to be saved.

**LIBRARY = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the program library in which the LLM is to be saved.

**ELEMENT =**

The element name and element version that the LLM is to receive on saving in the program library.

*Note*

It is not possible to store an LLM under the same name and with the same element version number if other tasks are simultaneously processing or including this LLM, since the library element is locked for write access in this case. In order to resolve this conflict, all but one of the competing tasks must temporarily store the LLM under another name or with a different version number.

**ELEMENT = \*CURRENT-NAME(...)**

BINDER takes the element name from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes as the element name of the LLM:

- the element name from the associated START-LLM-UPDATE statement if the LLM was modified by means of a START-LLM-UPDATE statement
- the internal name from the associated START-LLM-CREATION statement if the LLM was created by means of a START-LLM-CREATION statement.

**VERSION =**

Defines the element version.

**VERSION = \*CURRENT-VERSION**

BINDER takes the element version from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes as the version of the LLM:

- the specified element version from the associated START-LLM-UPDATE statement if the LLM was modified by means of a START-LLM-UPDATE statement,
- the internal version from the associated START-LLM-CREATION statement if the LLM was created by means of a START-LLM-CREATION statement.

**VERSION = \*INTERNAL-VERSION**

BINDER takes as the element version the internal version of the LLM.

**VERSION = \*UPPER-LIMIT**

BINDER takes as the element version the default value for the highest version in the case of program libraries (see the “LMS” manual [3]).



**VERSION = \*INCREMENT**

The current element version number is incremented by 1.

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**ELEMENT = \*INTERNAL-NAME(...)**

BINDER takes as the element name the internal name of the LLM.

**VERSION =**

Defines the element version.

**VERSION = \*INTERNAL-VERSION**

BINDER takes as the element version the internal version of the LLM.

**VERSION = \*UPPER-LIMIT**

BINDER takes as the element version the default value for the version in the case of program libraries (see the “LMS” manual [3]).

**VERSION = \*INCREMENT**

The current element version number is incremented by 1.

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**ELEMENT = <composed-name 1..64(...)**

Explicit specification of the element name.

Note: BINDER checks special data type <element-name> (see [page 199](#)). In DBL commands LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/ START-PROGRAM, see the “BLSSERV Binder Loader/Starter” manual [1]), the length of the element names is restricted to 32 characters when BLSSERV up to V2.4 is used. BINDER therefore issues the following warning if the element name is longer than 32 characters:

```
BND2110 WARNING: ELEMENT NAME LONGER THAN 32 CHARACTERS NOT PROCESSABLE BY
                'DBL'
```

**VERSION = \*INTERNAL-VERSION / \*UPPER-LIMIT / \*INCREMENT / <composed-name 1..24> / <c-string 1..24>**

Defines the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

See above for the meanings of the operands.

**ELEMENT = <c-string 1..64>(…)**

Explicit specification of the element name.

Note: BINDER checks special data type <element-name> (see [page 199](#)). In DBL commands LOAD-/START-EXECUTABLE-PROGRAM (or LOAD-/ START-PROGRAM, see the “BLSSERV Binder Loader/Starter” manual [1]), the length of the element names is restricted to 32 characters when BLSSERV up to V2.4 is used. BINDER therefore issues the following warning if the element name is longer than 32 characters:

```
BND2110 WARNING: ELEMENT NAME LONGER THAN 32 CHARACTERS NOT PROCESSABLE BY
'DBL'
```

**VERSION = \*INTERNAL-VERSION / \*UPPER-LIMIT / \*INCREMENT / <composed-name 1..24> / <c-string 1..24>**

Defines the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

See above for the meanings of the operands.

**MODULE-CONTAINER = \*FILE(…)**

The LLM is to be stored in a PAM file. This type of LLM is called a PAM-LLM.

*Notes*

- PAM-LLMs can be loaded as of BLSSERV V2.5.
- If an LLM which is stored in a PAM file is modified with START-LLM-UPDATE and saved into a PLAM library with SAVE-LLM, the value for \*CURRENT-NAME is the internal name of the LLM.
- When an LLM is output to a PAM file, BINDER initially generates a temporary PAM file and writes the LLM into it. Once the LLM has been successfully generated, it is written to the PAM file defined with SAVE-LLM and the temporary file is deleted. The BS2000 function “Temporary files” is used to create the temporary file if this is activated by the systems support staff with the class 2 system parameter TEMPFILE. Otherwise, BINDER generates a file whose name is made up of the name of the input file together with the date and time.

**FILE-NAME =**

Specifies the file in which the LLM is to be stored.

**FILE-NAME = \*CURRENT**

The LLM is stored in the PAM file specified in the most recent SAVE-LLM or START-LLM-UPDATE statement.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Name of the PAM file in which the LLM is to be stored.

**FILE-NAME = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the PAM file in which the PAM-LLM is to be stored.

**OVERWRITE =**

Specifies whether or not overwriting is permitted.

**OVERWRITE = \*LAST-SAVE**

BINDER uses the value from the last SAVE-LLM statement in this edit run. If this statement has not yet been entered in this edit run, BINDER uses the value \*STD.

**OVERWRITE = \*STD**

BINDER uses the value of the operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*YES.

**OVERWRITE = \*YES**

Overwriting is permitted.

**OVERWRITE = \*NO**

Overwriting is not permitted.

**FOR-BS2000-VERSIONS =**

Specifies the BS2000 OSD/BC version for which the generated LLM is to be loaded by DBL. DBL can only process the LLM in the specified (or a higher) version of BS2000 OSD/BC.

**FOR-BS2000-VERSIONS = \*BY-PROGRAM**

BINDER defines the format of the generated LLM on the basis of its contents. The LLM format is always the lowest possible that can provide the required functionality. For example, an LLM containing RISC code has format 3, while an LLM containing compressed text is created in format 2.

**FOR-BS2000-VERSIONS = \*LAST-SAVE**

BINDER uses the value from the last SAVE-LLM statement in this edit run. If this statement has not yet been entered in this edit run, BINDER uses the value \*STD.

**FOR-BS2000-VERSIONS = \*STD**

BINDER uses the value of the operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value \*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-CURRENT(...)**

The BS2000 OSD/BC version under which BINDER is currently running is used.

**CONNECTION-MODE =**

Defines the type of connection between the private and public slices. This operand is only meaningful if the LLM is divided into slices according to the public attribute of the CSECTs it contains.

**CONNECTION-MODE = \*OSD-DEFAULT**

This specification is supported for compatibility reasons. It is equivalent to CONNECTION-MODE = \*BY-RELOCATION

**CONNECTION-MODE = \*BY-RELOCATION**

The connection between private and public slices is by relocation.

**CONNECTION-MODE = \*BY-RESOLUTION**

The connection between private and public slices is by resolution.

**FOR-BS2000-VERSIONS = \*FROM-V10(...)**

The LLM can be loaded by DBL in any version of BS2000 OSD/BC.

**CONNECTION-MODE = \*BY-RESOLUTION / \*BY-RELOCATION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V1(...)**

The LLM can be loaded by DBL in any version of BS2000 OSD/BC.

**CONNECTION-MODE = \*BY-RELOCATION / \*BY-RESOLUTION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V3(...)**

The LLM can be loaded by DBL in BS2000/OSD-BC V3.0 or higher.

**CONNECTION-MODE = \*BY-RELOCATION / \*BY-RESOLUTION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**FOR-BS2000-VERSIONS = \*FROM-OSD-V4(...)**

The LLM can be loaded by BLSSERV in BS2000/OSD-BC V3.0 or higher.

**CONNECTION-MODE = \*BY-RELOCATION / \*BY-RESOLUTION**

The meaning of the operand and value is as for FOR-BS2000-VERSION=\*FROM-CURRENT.

**REQUIRED-COMPRESSION =**

Specifies whether the text information (TXT) is to be compressed for better utilization of the disk capacity.

**REQUIRED-COMPRESSION = \*LAST-SAVE**

BINDER uses the value of the REQUIRED-COMPRESSION operand in the last SAVE-LLM statement in this edit run. If this statement has not yet been entered in this edit run, BINDER uses the value \*STD.

**REQUIRED-COMPRESSION = \*STD**

BINDER uses the value specified in the last MODIFY-STD-DEFAULTS statement. If this statement has not yet been entered in this edit run, BINDER uses the value NO.

**REQUIRED-COMPRESSION = \*NO**

The text information is not compressed.

**REQUIRED-COMPRESSION = \*YES**

The text information is compressed.

**NAME-COLLISION =**

Specifies how name conflicts which occur during processing of the SAVE-LLM statement are to be handled.

**NAME-COLLISION = \*LAST-SAVE**

BINDER uses the value from the last SAVE-LLM statement in this edit run. If this statement has not already been entered in the same edit run, BINDER uses the value STD.

**NAME-COLLISION = \*STD**

BINDER uses the value from the NAME-COLLISION operand in the last MODIFY-STD-DEFAULTS statement. If this statement has not already been entered in the same edit run, BINDER uses the value IGNORED.

**NAME-COLLISION = \*IGNORED**

Name conflicts are not handled.

**NAME-COLLISION = \*WARNING(...)**

The user receives a warning if name conflicts occur during processing of the SAVE-LLM statement.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**NAME-COLLISION = \*ERROR(...)**

Execution of the SAVE-LLM statement is aborted if name conflicts (correctable errors) occur.

**SCOPE =**

Defines the scope of the definitions for handling name conflicts.

**SCOPE = \*WHOLE-LLM**

The definitions for name conflicts are valid for the entire LLM.

**SCOPE = \*SLICE**

The definitions for name conflicts are valid only at the slice level, i.e. name conflicts between different slices are ignored by BINDER.

This value may be specified only for user-defined slices.

**SYMBOL-DICTIONARY =**

Specifies whether the LLM is saved with or without an External Symbols Vector (ESV).

*Note*

If LOGICAL-STRUCTURE=\*NONE has been set, whether explicitly or implicitly (by means of \*LAST-SAVE), SYMBOL-DICTIONARY=\*NO must be set.

**SYMBOL-DICTIONARY = \*LAST-SAVE**

BINDER takes the values from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes the value YES.

**SYMBOL-DICTIONARY = \*YES**

The External Symbols Vector (ESV) is taken over on saving the LLM.

**SYMBOL-DICTIONARY = \*NO**

No External Symbols Vector (ESV) is taken over on saving the LLM.

**RELOCATION-DATA =**

Specifies whether the LLM is saved with or without relocation information.

**RELOCATION-DATA = \*LAST-SAVE**

BINDER takes the values from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes the value YES.

**RELOCATION-DATA = \*YES**

The relocation information is taken over on saving the LLM.

**RELOCATION-DATA = \*NO**

No relocation information is taken over on saving the LLM. The value NO is skipped in the case of slices by attributes.

**RELOCATION-DATA = \*UNRESOLVED-ONLY**

Only the relocation information for external references which are still unresolved is stored.

**LOGICAL-STRUCTURE =**

Specifies whether the LLM is saved with or without logical structure information.

**LOGICAL-STRUCTURE = \*LAST-SAVE**

BINDER takes the values from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes the value WHOLE-LLM.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over on saving the LLM. The logical structure of the LLM can then be modified subsequently.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

On saving the LLM, a structure comprising only the internal name (root) and object modules (OMs) is stored.

**LOGICAL-STRUCTURE = \*NONE**

Only the internal name (root) of the structure is stored.

**TEST-SUPPORT =**

Specifies whether the LLM is saved with or without LSD information.



If LOGICAL-STRUCTURE=\*NONE or SYMBOL-DICTIONARY=\*NO has been set, whether explicitly or implicitly (by means of \*LAST-SAVE), TEST-SUPPORT=\*NO must be set.

Information on the existence of the list for symbolic debugging is provided in the "T&D" column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*LAST-SAVE**

BINDER takes the values from the most recent SAVE-LLM statement in the same edit run. If no SAVE-LLM statement has as yet been specified in the edit run, BINDER takes the value YES.

**TEST-SUPPORT = \*YES**

The LSD information is taken over on saving the LLM.

**TEST-SUPPORT = \*NO**

No LSD information is taken over on saving the LLM.

**LOAD-ADDRESS =**

Defines a virtual address at which the LLM is to be loaded. This operand is skipped in the case of slices by attributes.

**LOAD-ADDRESS = \*LAST-SAVE**

BINDER takes the address from the most recent SAVE-LLM statement in the same edit run. If no corresponding SAVE-LLM statement has as yet been specified in the edit run, the following value is used for the load address:

- UNDEFINED if the LLM was created with a START-LLM-CREATION statement
- the original load address if the LLM was updated by means of a START-LLM-UPDATE statement.

**LOAD-ADDRESS = \*UNDEFINED**

The virtual address is set as follows:

- to the value 0 if the RELOCATION-DATA operand has the value \*NO or \*UNRESOLVED-ONLY
- above 16 Mbyte to the value of the class 2 system option BLSLDPXS if this was defined during system installation
- to the value X'FFFFFFFF' in all other cases.

**LOAD-ADDRESS = NULL**

The virtual address 0 is defined as the load address.



**LOAD-ADDRESS = <x-string 1..8>**

Explicit specification of the load address.

The address must lie on a page boundary, i.e. be a multiple of 4096 (X'1000'). Addresses not lying on a page boundary are automatically aligned on a page boundary by BINDER.

**LOAD-ADDRESS = \*BY-SLICES(...)**

Specification of the load addresses for individual slices. Load addresses of slices which are not explicitly specified are calculated by BINDER. For LLMs which are not output in PAM files, this specification is handled in the same way as \*UNDEFINED.

**ADDRESSES = list-poss(40): \*REGION(...) / \*SLICE(...)**

Specifies the load addresses for slices.

**\*REGION(...)**

Defines the load address for a region.

**REGION-NAME = <structured-name 1..32>**

Name of the region for which a load address is defined.

**REGION-ADDRESS = <x-string 1..8>**

Load address for the region.

**\*SLICE(...)**

Defines the load address for a slice.

**SLICE-NAME = \*ROOT / <structured-name 1..32>**

Name of the slice for which a load address is defined.

**SLICE-ADDRESS = <x-string 1..8> / \*BEHIND-SLICE(...)**

Load address for the slice.

**\*BEHIND-SLICE(...)****SLICE = <structured-name 1..32>**

Name of a slice that is used to compute the load address of the slice specified in SLICE-NAME.

*Note*

If the user has defined a load address for a slice, it is accepted by BINDER only if it is at least as long as the smallest possible load address permitted by the physical structure of the LLM.

*Example*

Slice X has the load address X'1000' and a length of X'4000'. Slice Y links itself to slice X. The smallest possible load address for slice Y is thus X'4000' + X'1000' = x'5000'. A user-defined slice address is therefore accepted by BINDER only if it is x'5000' or larger. If a smaller load address is specified, BINDER replaces it with X'5000'.

**ENTRY-POINT =**

Specifies the name of a symbol, i.e. the name of an address to be branched to after loading of the LLM.

**ENTRY-POINT = \*LAST-SAVE**

BINDER takes the name of the symbol from the most recent SAVE-LLM statement in the same edit run.

If no corresponding SAVE-LLM statement has as yet been specified in the edit run, the following name is defined for the symbol:

- the name from the END record of the *first* object module if the LLM was created with a START-LLM-CREATION statement
- the original name if the LLM was updated with a START-LLM-UPDATE statement.

**ENTRY-POINT = \*STD**

The name of the symbol is taken over from the END record of the first object module (OM). If the END record contains no such specification, then, on loading, a branch is taken to the address defined by the *first* byte of the first object module.

**ENTRY-POINT = \*BY-MODULE(...)**

Defines the object module from whose END record the name of the symbol is to be taken over. If the END record contains no such specification, then, on loading, a branch is taken to the address defined by the *first* byte of the object module.

**PATH-NAME = <text 1..255>**

Path name of the object module (see [page 24ff](#)).

If the path name of a sub-LLM is specified, the *first* object module of the sub-LLM is assumed.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

**ENTRY-POINT = <text 1..32>**

Explicit specification of the name of a CSECT or of an ENTRY for the name of the symbol.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**MAP = \*LAST-SAVE / \*YES / \*NO**

Specifies whether lists containing information about the saved LLM are output (see [page 141ff](#)).

The output destination and the type of lists are determined by the values in the most recent preceding MODIFY-MAP-DEFAULTS statement with the specification MAP-NAME=\*STD, or without a MAP-NAME specification. If no MODIFY-MAP-DEFAULTS statement has been specified with MAP-NAME=\*STD or without the MAP-NAME operand, the default values are assumed.

**MAP = \*LAST-SAVE**

BINDER uses the value from the last SAVE-LLM statement in this edit run. If this statement has not yet been entered in this run, BINDER uses the value \*YES.

## SET-EXTERN-RESOLUTION

This statement defines how BINDER is to handle outstanding external references that cannot be resolved. It is possible to specify that unresolved external references are to be allowed or not allowed.

If unresolved external references are allowed, they are taken over on saving the LLM. Here the unresolved external references can be given a specified address. If unresolved external references are not allowed, the LLM will be rejected on saving with the SAVE-LLM statement.

### SET-EXTERN-RESOLUTION

```

SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
, SYMBOL-TYPE = *REFERENCES / list-poss(3): *EXTRN / *VCON / *WXTRN
, SCOPE = *CURRENT-SUB-LLM / *EXPLICIT(...) / *WHOLE-LLM
  *EXPLICIT(...)
    | WITHIN-SUB-LLM = *WHOLE-LLM / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
    | EXCEPT-SUB-LLM = *NONE / list-poss(10): <c-string 1..255 with-low> / <text 1..255>
, RESOLUTION = *STD / *BY-SYMBOL(...) / *MANDATORY
  *BY-SYMBOL(...)
    | SYMBOL = <c-string 1..32 with-low> / <text 1..32>

```

### **SYMBOL-NAME =**

Defines the external references that BINDER is to handle if they cannot be resolved.

### **SYMBOL-NAME = \*ALL**

All external references are to be handled.

### **SYMBOL-NAME = list-poss(40): <c-string 1..255> / <text 1..32>**

Names of the external references to be handled. Wildcards may be specified.

Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

### **SYMBOL-TYPE = \*REFERENCES / list-poss(3): \*EXTRN / \*VCON / \*WXTRN**

Defines the type of the external references to be handled. External references (EXTRNs), V-type constants (VCON) and weak external references (WXTRNs) can be selected. If REFERENCES is specified, all types of external references are handled.

**SCOPE =**

Sets one or more pointers. These point to the sub-LLMs in the logical structure of the LLM in which BINDER is to handle unresolved external references.

**SCOPE = \*CURRENT-SUB-LLM**

The pointer points to the current sub-LLM (see BEGIN-SUB-LLM-STATEMENTS statement).

**SCOPE = \*EXPLICIT(...)****WITHIN-SUB-LLM = \*WHOLE-LLM / list-poss(10): <text 1..255>**

The pointers point to the explicitly specified sub-LLMs. The path names of the sub-LLMs should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*WHOLE-LLM is specified, all sub-LLMs are referenced.

**EXCEPT-SUB-LLM = \*NONE / list-poss(10): <text 1..255>**

Allows the exclusion of individual pointers from the list specified in the WITHIN-SUB-LLM operand. The path names of the sub-LLMs that are to be ignored should be specified for <text 1..255>.

Note: BINDER checks special data type <path-name> (see [page 199](#)).

When \*NONE is specified, no pointers are excluded.

**SCOPE = \*WHOLE-LLM**

All sub-LLMs of the current LLM are involved.

**RESOLUTION =**

Defines the way in which BINDER is to handle the unresolved external references.

**RESOLUTION = \*STD**

Unresolved external references are allowed. When the LLM is saved with the SAVE-LLM statement the unresolved external references are taken over.

**RESOLUTION = \*BY-SYMBOL(...)**

Specifies that unresolved external references are given an address.

**SYMBOL = <c-string 1..32 with-low> / <text 1..32>**

Explicit specification of the address.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**RESOLUTION = \*MANDATORY**

Unresolved external references are not allowed. The LLM will be rejected on saving with the SAVE-LLM statement if unresolved external references are present.



**SLICE = \*CURRENT-SLICE**

The slice is added immediately following the current slice.

**SLICE = \*ROOT**

The slice is added immediately following the root slice (%ROOT).

**SLICE = <structured-name 1..32>**

Explicit specification of the slice, immediately following which the slice is added.

**POSITION = \*BEGIN-REGION(...)**

Defines the beginning of a region, i.e. ensures that the slice beginning there does not overlay preceding slices.

**REGION =**

Name of the region at which the slice is to begin.

**REGION = \*CURRENT-REGION**

Current region

The current region contains the current slice. If the current slice lies in the region containing the root slice (%ROOT), the name of the region must be specified explicitly.

**REGION = <structured-name 1..32>**

Explicit specification of the region.

The name of the region must be unique within the physical structure.

**NEW-REGION = \*NO / \*YES**

Specifies whether the specified region is new.

**MODE = \*UPDATE**

The slice is already present in the physical structure.

## SHOW-DEFAULTS

This statement permits the user to display the default values.  
The values are output on SYSOUT.

SHOW-DEFAULTS

```

STD-DEFAULTS = *YES / *NO
, CURRENT-DEFAULTS = *YES / *NO
, INCLUSION-DEFAULTS = *YES / *NO
, LAST-SAVE = *YES / *NO
, MAP-DEFAULTS = *YES (...) / *NO
  *YES(...)
    | MAP-NAME = *STD / *ALL / <structured-name 1..32>

```

**STD-DEFAULTS = \*YES / \*NO**

Displays all defaults specified with the operand value STD.

**CURRENT-DEFAULTS = \*YES / \*NO**

Displays all defaults which apply to CURRENT operands.

**INCLUSION-DEFAULTS = \*YES / \*NO**

Displays all defaults which apply to the inclusion of modules.

**LAST-SAVE = \*YES / \*NO**

Displays all defaults which were defined the last time an LLM was saved.

**MAP-DEFAULTS = \*YES(...) / \*NO**

Specifies whether the defaults for list output are to be displayed.

**MAP-DEFAULTS = \*YES(...)**

The defaults for list output are displayed.

**MAP-NAME =**

Specifies the name of the list whose defaults are to be displayed.

**MAP-NAME = \*STD**

The defaults for the list with the default name `BNDMAP.date.time.<tsn>` are to be displayed.

**MAP-NAME = \*ALL**

The defaults for the list with the default name and the defaults for all self-defined lists are to be displayed.

**MAP-NAME = <structured-name 1..32>**

The defaults for the self-defined list with the specified name are to be displayed.



## SHOW-LIBRARY-ELEMENTS

This statement permits the user to obtain information about library elements (object modules and LLMs) during a BINDER run. The user can also check whether the symbols in the object modules and/or LLMs could result in name conflicts. By default, the information is output on SYSLST, but the user can specify other output devices. Except for the list which is output on SYSLST, the lists are ISAM files with ISAM keys with a length of 8 (see also [page 141](#)).

### SHOW-LIBRARY-ELEMENTS

```

LIBRARY = *CURRENT-INPUT-LIB / *BLSLIB-LINK /
           list-poss(40): <filename 1..54 without-gen-vers> / *LINK(...)
           *LINK(...)
             | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
ELEMENT = *ALL (...) / list-poss(40): <composed-name 1..64>(…) / <c-string 1..64>(…)
           *ALL(…)
             | VERSION = *ALL / *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
             <composed-name>(…)
             | VERSION = *ALL / *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
             <c-string>(…)
             | VERSION = *ALL / *HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>
,TYPE = (*L,*R) / list-poss(2): *L / *R
,SYMBOL-NAME = *ALL / *NONE / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
,SYMBOL-TYPE = (*CSECT,*ENTRY) / list-poss(2): *CSECT / *ENTRY
,SELECT = *ALL / *NAME-COLLISION

```

continued ➡

```

,OUTPUT = *SYSLST (...) / *BY-SHOW-FILE(...) / <filename 1..54 without-gen-vers>(…) / *LINK(...) /
          *EXIT-ROUTINE(...)

*SYSLST(...)
  |
  | SYSLST-NUMBER = STD / <integer 1..99>
  |
  | ,LINES-PER-PAGE = 64 / <integer 10..2147483647> / *IGNORED
  |
  | ,LINE-SIZE = 72 / <integer 72..255>
  |
*BY-SHOW-FILE(...)
  |
  | FILE-NAME = *STD / <filename 1..54 without-gen-vers>
  |
  | ,DELETE-FILE = *YES / *NO
  |
  | ,LINE-SIZE = 72 / <integer 72..255>
  |
<filename>(…)
  |
  | LINE-SIZE = 72 / <integer 72..255>
  |
*LINK(...)
  |
  | LINK-NAME = BNDMAP / <structured-name 1..8> / <filename 1..8 without-gen>
  |
  | ,LINE-SIZE = 72 / <integer 72..255>
  |
*EXIT-ROUTINE(...)
  |
  | ROUTINE-NAME =
  |
  | <c-string 1..32 with-low> / <text 1..32>
  |
  | ,LIBRARY = *BLSLIB-LINK / <filename 1..54 without-gen-vers> / *LINK(...)
  |
  | *LINK(...)
  |
  | | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
  |
  | ,FILE-NAME = *STD / <filename 1..54 without-gen-vers>
  |
  | ,LINE-SIZE = 72 / <integer 72..255>
  |
  | ,USER-PARAMETERS = *NONE / <c-string 1..255 with-low> / <text 1..255>

```

**LIBRARY =**

Specifies the library or libraries to be searched. This operand is mandatory.

**LIBRARY = \*CURRENT-INPUT-LIB**

The library to be searched is the one from which the last element was read (with a START-LLM-UPDATE, INCLUDE-MODULES or REPLACE-MODULES statement). The scope of the operand relates to one edit run.

**LIBRARY = \*BLSLIB-LINK**

The libraries with the file link names BLSLIBnn (00≤nn≤99) are to be searched. The libraries are searched in *ascending* order of the values “nn” in the file link name.

**LIBRARY = list-poss(40): <filename 1..54 without-gen-vers>**

The file name of the library which is to be searched.

**LIBRARY = \*LINK(...)**

Denotes a library with the file link name

**LINKNAME = <structured-name 1..8> / <filename 1..8 without-gen>**

File link name of the library that is to be searched.

**ELEMENT =**

Specifies the element name and the element version of the modules which are to be checked.

**ELEMENT = \*ALL(...)**

All elements of the specified library are to be checked. The elements are processed in the order in which they are stored in the library.

**VERSION =**

Specifies the element version of the module. The element version is valid only for program libraries.

**VERSION = \*ALL**

All versions of library elements with the same name are to be checked.

**VERSION = \*HIGHEST-EXISTING**

BINDER takes as element version the default value for the highest version in the case of program libraries (see the "LMS" manual [3]).

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**ELEMENT = <composed-name 1..64>(...)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)).

**VERSION = \*ALL / \*HIGHEST-EXISTING / <composed-name 1..24> / <c-string 1..24>**

Specifies the element version of the module. The element version is valid only for program libraries.

See above for the meanings of the operands.

**ELEMENT = <c-string 1..64>(…)**

Explicit specification of the element name and element version.

Note: BINDER checks special data types <element-name> and <element-version> (see [page 199](#)). See above for the meanings of the operands.

**TYPE =**

Defines the priority of the modules (object modules and/or LLMs) to be checked.

**TYPE = (\*L,\*R)**

Both LLMs and object modules are to be checked. If an LLM and an object module have the same name, the *LLM* is checked.

**TYPE = (\*R,\*L)**

Both LLMs and object modules are to be checked. If an LLM and an object module have the same name, *object module* is checked.

**TYPE = \*R**

Only object modules are checked.

**TYPE = \*L**

Only LLMs are checked.

**SYMBOL-NAME =**

Specifies the symbols in the library element to be processed.

**SYMBOL-NAME = \*ALL**

All symbols are processed.

**SYMBOL-NAME = \*NONE**

No symbols are processed. When the library contents are output, only LLMs (without the related symbols) will appear.

**SYMBOL-NAME = list-poss(40): <c-string 1..255> / <text 1..32>**

Specifies the names of the symbols which are to be processed. Wildcards may be specified. Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

**SYMBOL-TYPE = (\*CSECT,ENTRY\*) / list-poss(2): \*CSECT / \*ENTRY**

Specifies which types of symbols are to be processed. A list of CSECTs and ENTRYs, a list of CSECTs or a list of ENTRYs can be requested.

**SELECT =**

Specifies what is to be done with the symbols specified for SYMBOL-NAME.

**SELECT = \*ALL**

All symbols specified for SYMBOL-NAME are listed.

**SELECT = \*NAME-COLLISION**

From the symbols specified for SYMBOL-NAME, the program selects those symbols which could cause a name conflict if the element being considered is included in the LLM currently being processed.

BINDER searches for duplicates also in the LLM currently in process.

If a duplicate is detected in the LLM, the following three lines are output:

- "IN THE LLM <LLM name> CURRENTLY IN PROCESS".
- Type and name of the object that owns the symbol.
- Type of the symbol in the object.

**OUTPUT =**

Specifies the output destination for the lists.

**OUTPUT = \*SYSLST(...)**

The output destination is a system file SYSLST.

**SYSLST-NUMBER =****SYSLST-NUMBER = \*STD**

The system file SYSLST is used.

**SYSLST-NUMBER = <integer 1..99>**

A system file from the set SYSLST01 to SYSLST99, whose number must be specified here, is used.

**LINES-PER-PAGE = 64 / <integer 10..2147483647> / IGNORED**

Specifies the number of lines per page. This is needed for generation of the form-feeds at the end of each page. If LINES-PER-PAGE=IGNORED is specified, *no* form-feed is executed.

**LINE-SIZE = 72 / <integer 72..255>**

Specifies the number of characters per line.

**OUTPUT = \*BY-SHOW-FILE(...)**

The output destination is an ISAM file whose file name is specified here. After output, the file is automatically opened with the command SHOW-FILE (see the "Commands" manual [5]). The ISAM keys contained in the file are described in the appendix (see [page 369f](#)).

**FILE-NAME =**

Specifies the name for the ISAM file.

**FILE-NAME = \*STD**

The output is sent to the file with the default file name

BNDMAP.date.time.<tsn>

“date” has the format yyyy-mm-ddoy

yyyy        year

mm         month

dd         day

doy        day of year

“time” has the format hhmmss

hh         hours

mm         minutes

ss         seconds

“<tsn>” means the TSN (Task Sequence Number) of the current task.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Explicit specification of the file name.

**DELETE-FILE = \*YES / NO**

Specifies whether the file is to be deleted after execution of the SHOW-FILE command.

**LINE-SIZE = 72 / <integer 72..255>**

Specifies the number of characters per line.

**OUTPUT = <filename 1..54 without-gen-vers>**

The output destination is an ISAM file whose file name is specified here. The ISAM keys contained there are described in the appendix (see [page 369f](#)).

**LINE-SIZE = 72 / <integer 72..255>**

Specifies the number of characters per line.

**OUTPUT = \*LINK(...)**

The output destination is an ISAM file whose link name is specified here. The ISAM keys contained there are described in the appendix (see [page 369f](#)).

**LINK-NAME = BNDMAP / <structured-name 1..8> / <filename 1..8 without-gen>**

Specifies the file link name. The default file link name is BNDMAP.

**LINE-SIZE = 72 / <integer 72..255>**

Specifies the number of characters per line.

**OUTPUT = \*EXIT-ROUTINE(...)**

The list is output to the ISAM file specified for FILE-NAME and control is then passed to the subroutine by calling the BIND macro. See the SHOW-MAP statement for register conventions and for the parameters for the BIND macro.

**ROUTINE-NAME = <text 1..32>**

The name of the subroutine which is to be called.

**LIBRARY =**

Permits specification of the library which contains the subroutine.

**LIBRARY = \*BLSLIB-LINK**

Libraries with the file link names BLSLIBnn (00≤nn≤99) are used. These libraries are searched in *ascending* order of the “nn” values in the file link name.

**LIBRARY = <filename 1..54 without-gen-vers>**

Explicit specification of the library name.

**LIBRARY = \*LINK(...)**

The library is specified with its file link name.

**LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>**

The file link name of the library.

**FILE-NAME = \*STD / <filename 1..54 without-gen-vers>**

Specifies the name (or the link name) of the ISAM file to which the list is to be output. By default, the file name BNDMAP.date.time.<tsn> is used (see the OUTPUT=\*BY-SHOW-FILE(...) operand for the format). The ISAM keys contained in the file are described in the appendix (see [page 369f](#)).

**LINE-SIZE = 72 / <integer 72..255>**

Specifies the number of characters per line.

**USER-PARAMETERS = \*NONE / <c-string 1..255> / <text 1..255>**

Specifies the parameters to be passed with the BINDER macro.

## SHOW-MAP

This statement outputs lists containing information about the current LLM (see [page 141ff](#)). The **\*MAP-DEFAULT** value is the value defined with a previous **MODIFY-MAP-DEFAULTS** statement with the same **MAP-NAME** specification for the same operand. If a **MODIFY-MAP-DEFAULTS** statement with the same **MAP-NAME** specification has not yet been entered, the first operand value after the **MAP-DEFAULT** value is used.

As well as the standard list **\*STD**, you can also view other (named) lists for which you have defined default values with the statement

**MODIFY-MAP-DEFAULTS MAP-NAME= <structured-name 1..32>**.

### SHOW-MAP

```

MAP-NAME = *STD / <structured-name 1..32>
, USER-COMMENT = *MAP-DEFAULT / *NONE / <c-string 1..255 with-low>
, HELP-INFORMATION = *MAP-DEFAULT / *YES / *NO
, GLOBAL-INFORMATION = *MAP-DEFAULT / *YES / *NO
, LOGICAL-STRUCTURE = *MAP-DEFAULT / *YES(...) / *NO
  *YES(...)
    | RESOLUTION-SCOPE = *MAP-DEFAULT / *YES / *NO
    | ,HSI-CODE = *MAP-DEFAULT / *YES / *NO / *X86
, PHYSICAL-STRUCTURE = *MAP-DEFAULT / *YES / *NO
, PROGRAM-MAP = *MAP-DEFAULT / *PARAMETERS(...) / *NO
  *PARAMETERS(...)
    | DEFINITIONS = *MAP-DEFAULT / *ALL / *NONE / list-poss(5): *MODULE / *CSECT /
      *ENTRY / *COMMON / *XDSECT-D
    | ,INVERTED-XREF-LIST = *MAP-DEFAULT / *NONE / *ALL / list-poss(4): *EXTRN / *VCON /
      *WXTRN / *XDSECT-R
    | ,REFERENCES = *MAP-DEFAULT / *ALL / *NONE / list-poss(4): *EXTRN / *VCON /
      *WXTRN / *XDSECT-R

```

continued ➔



```

,UNRESOLVED-LIST = *MAP-DEFAULT / *SORTED(...) / *YES(...) / *NO
  *SORTED(...)
    | WXTRN = *YES / *NO
    | ,NOREF = *NO / *YES
  *YES(...)
    | WXTRN = *YES / *NO
    | ,NOREF = *NO / *YES
,SORTED-PROGRAM-MAP = *MAP-DEFAULT / *NO / *YES
,PSEUDO-REGISTER = *MAP-DEFAULT / *NO / *YES
,UNUSED-MODULE-LIST = *MAP-DEFAULT / *NO / *YES
,DUPLICATE-LIST = *MAP-DEFAULT / *NO / *YES(...)
  *YES(...)
    | INVERTED-XREF-LIST = *YES / *NO
,MERGED-MODULES = *MAP-DEFAULT / *YES / *NO
,INPUT-INFORMATION = *MAP-DEFAULT / *YES / *NO
,STATEMENT-LIST = *MAP-DEFAULT / *NO / *YES
,OUTPUT = *MAP-DEFAULT / *SYSLST(...) / *BY-SHOW-FILE(...) / <filename 1..54 without-gen-vers>(...) /
  *LINK(...) / *EXIT-ROUTINE(...)
  *SYSLST(...)
    | SYSLST-NUMBER = *STD / <integer 1..99>
    | ,LINES-PER-PAGE = 64 / <integer 10..2147483647> / *IGNORED
    | ,LINE-SIZE = 136 / <integer 132..255>
  *BY-SHOW-FILE(...)
    | FILE-NAME = *STD / <filename 1..54 without-gen-vers>
    | ,DELETE-FILE = *YES / *NO
    | ,LINE-SIZE = 136 / <integer 132..255>
  <filename 1..54 without-gen-vers>(...)
    | LINE-SIZE = 136 / <integer 132..255>
  *LINK(...)
    | LINK-NAME = BNDMAP / <structured-name 1..8> / <filename 1..8 without-gen>
    | ,LINE-SIZE = 136 / <integer 132..255>

```

continued ➡

```

*EXIT-ROUTINE(...)
  |
  | ROUTINE-NAME = <c-string 1..32 with-low> / <text 1..32>
  |
  | ,LIBRARY = *BLSLIB-LINK / <filename 1..54 without-gen-vers> / *LINK(...)
  |
  |   *LINK(...)
  |   |
  |   | LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>
  |   |
  |   | ,FILE-NAME = *STD / <filename 1..54 without-gen-vers>
  |   |
  |   | ,LINE-SIZE = 136 / <integer 132..255>
  |   |
  |   | ,USER-PARAMETERS = *NONE / <c-string 1..255 with-low> / <text 1..255>
  |
  |

```

**MAP-NAME =**

Specifies the name of the list to be output.

**MAP-NAME = \*STD**

The default list with the name `BNDMAP.date.time.<tsn>` is output.

**MAP-NAME = <structured-name 1..32>**

A list defined by the user with the `MODIFY-MAP-DEFAULTS` statement is output.

**USER-COMMENT = \*MAP-DEFAULT / \*NONE / <c-string 1..255 with-low>**

Specifies whether a header line with a user comment is output in the lists. The header line is repeated at the beginning of each list.

**HELP-INFORMATION = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether a list of abbreviations is output.

This list explains the abbreviations used in the subsequent lists (e.g. SD for CSECT, GM for prelinked module).

**GLOBAL-INFORMATION = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether the header information is also to be output.

**LOGICAL-STRUCTURE = \*MAP-DEFAULT / \*YES(...) / \*NO**

Specifies whether a list mapping the logical structure of the LLM is logged.

**LOGICAL-STRUCTURE = \*YES(...)**

Logs the logical structure of the LLM.

**RESOLUTION-SCOPE = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether the output is to contain the priority classes assigned to the logical node of the LLM.

**HSI-CODE = \*MAP-DEFAULT / \*YES / \*NO / \*X86**

Specifies whether the output contains the HSI code specification.

**PHYSICAL-STRUCTURE = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether a list mapping the physical structure of the LLM is logged.

**PROGRAM-MAP = \*MAP-DEFAULT / \*PARAMETERS(...) / \*NO**

Specifies whether a program map is logged.

*Note*

The program overview is always output with module information when UNUSED-MODULE-LIST=\*YES is set, regardless of the value of the PROGRAM-MAP operand.

**PROGRAM-MAP = \*PARAMETERS(...)**

Defines the contents of the program map.

**DEFINITIONS = \*MAP-DEFAULT / \*ALL / \*NONE / list-poss(5): \*MODULE / \*CSECT / \*ENTRY / \*COMMON / \*XDSECT-D**

Defines which program definitions the program map will contain.

**DEFINITIONS = \*ALL**

The program map contains all the program definitions listed below.

**DEFINITIONS = \*NONE**

The program map contains no program definitions.

**DEFINITIONS = \*MODULE**

The program map contains module information.

**DEFINITIONS = \*CSECT**

The program map contains definitions of CSECTs.

**DEFINITIONS = \*ENTRY**

The program map contains definitions of ENTRYs.

**DEFINITIONS = \*COMMON**

The program map contains definitions of COMMONs.

**DEFINITIONS = \*XDSECT-D**

The program map contains definitions of external dummy sections.

**INVERTED-XREF-LIST = \*MAP-DEFAULT / \*NONE / \*ALL / list-poss(4): \*EXTRN / \*VCON / \*WXTRN / \*XDSECT-R**

Defines the contents of a cross-reference list that contains the resolved references with cross-references to the associated program definitions for each module.

**INVERTED-XREF-LIST = \*NONE**

No cross-reference list is output.

**INVERTED-XREF-LIST = \*ALL**

The cross-reference list contains all the resolved references listed below for each module.

**INVERTED-XREF-LIST = \*EXTRN**

The cross-reference list contains the resolved EXTRNs for each module.

**INVERTED-XREF-LIST = \*VCON**

The cross-reference list contains the resolved V-type constants for each module.

**INVERTED-XREF-LIST = \*WXTRN**

The cross-reference list contains the resolved weak external references for each module.

**INVERTED-XREF-LIST = \*XDSECT-R**

The cross-reference list contains the resolved dummy section references for each module.

**REFERENCES = \*MAP-DEFAULT / \*ALL / \*NONE /****list-poss(4): \*EXTRN / \*VCON / \*WXTRN / \*XDSECT-R**

Specifies which reference list the program map is to contain. See INVERTED-XREF-LIST operand for meaning of operand values.

**UNRESOLVED-LIST = \*MAP-DEFAULT / \*SORTED(...) / \*YES(...) / \*NO**

Specifies whether or not a list of the unresolved external references is logged and defines its contents.

**UNRESOLVED-LIST = \*SORTED(...)**

Unresolved external references are output sorted.

**WXTRN = \*YES / \*NO**

Specifies whether or not weak external references are also to be output.

**NOREF = \*NO / \*YES**

Specifies whether or not unresolved external references are also to be output.

**UNRESOLVED-LIST = \*YES(...)**

The unresolved external references are output in the order in which they are found.

**WXTRN = \*YES / \*NO**

Specifies whether or not weak external references are also to be output.

**NOREF = \*NO / \*YES**

Specifies whether or not unresolved external references are also to be output.

**UNRESOLVED-LIST = \*NO**

Unresolved external references are not logged.

**SORTED-PROGRAM-MAP = \*MAP-DEFAULT / \*NO / \*YES**

Specifies whether a sorted list of program definitions is output.

**PSEUDO-REGISTER = \*MAP-DEFAULT / \*NO / \*YES**

Specifies whether a sorted list of the pseudo-registers is output.

**UNUSED-MODULE-LIST = \*MAP-DEFAULT / \*NO / \*YES**

Specifies whether a list of the unused modules is output.

**DUPLICATE-LIST = \*MAP-DEFAULT / \*NO / \*YES(...)**

Specifies whether a sorted list of the duplicate program definitions is output.

**DUPLICATE-LIST = \*YES(...)**

A list of the duplicate program definitions is output.

**INVERTED-XREF-LIST = \*YES / \*NO**

Specifies whether cross-references are output in the list of duplicate program definitions.

**MERGED-MODULES = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether merged modules are to be included in the list.

**INPUT-INFORMATION = \*MAP-DEFAULT / \*YES / \*NO**

Specifies whether a list containing input information about the LLM is output.

**STATEMENT-LIST = \*MAP-DEFAULT / \*NO / \*YES**

Defines whether a list of the recorded BINDER statements is output (see //START-STATEMENT-RECORDING and //STOP-STATEMENT-RECORDING). If a recorded statement was not terminated correctly, it could be that recording of this statement is incomplete.

**OUTPUT = \*MAP-DEFAULT / \*SYSLST(...) / \*BY-SHOW-FILE(...) / <filename 1..54 without-gen-vers> / \*LINK(...)**

Defines the output destination for the lists.

**OUTPUT = \*SYSLST(...)**

The output destination is a system file SYSLST.

**SYSLST-NUMBER =**

**SYSLST-NUMBER = \*STD**

The system file SYSLST is the output destination.

**SYSLST-NUMBER = <integer 1..99>**

One of the system files SYSLST01 through SYSLST99 whose number is specified here is the output destination.

**LINES-PER-PAGE = 64 / <integer 10..2147483647>**

Defines the number of lines per page.

**LINE-SIZE = 136 / <integer 132..255>**

Defines the number of characters per line.

**OUTPUT = \*BY-SHOW-FILE(...)**

The output destination is an ISAM file defined by its file name. The file is subsequently opened automatically by the SHOW-FILE command (see the “Commands” manual [5]). The file contains ISAM keys that are described in the appendix (see [page 369f](#)).

**FILE-NAME =**

Defines the file name of the file.

**FILE-NAME = \*STD**

Logging takes place in the file with the default file name

BNDMAP.date.time.<tsn>

“date” has the format yyyy-mm-ddoy

yyyy      year

mm        month

dd        day

doy       day of year

“time” has the format hhmmss

hh        hours

mm        minutes

ss        seconds

“tsn” signifies the TSN (Task Sequence Number) of the current task.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Explicit specification of the file name.

**DELETE-FILE = \*YES / \*NO**

Specifies whether the file is to be deleted after execution of the SHOW-FILE command.

**LINE-SIZE = 136 / <integer 132..255>**

Defines the number of characters per line.

**OUTPUT = <filename 1..54 without-gen-vers>**

The output destination is an ISAM file defined by the specified file name. The ISAM keys contained there are described in the appendix (see [page 369f](#)).

**LINE-SIZE = 136 / <integer 132...255>**

Defines the number of characters per line.

**OUTPUT = \*LINK(...)**

The output destination is an ISAM file defined by the file link name. The ISAM keys contained there are described in the appendix (see [page 369f](#)).

**LINK-NAME = BNDMAP / <filename 1..8 without-gen>**

Defines the file link name. The default file link name is BNDMAP.

**LINE-SIZE = 136 / <integer 132..255>**

Defines the number of characters per line.

**OUTPUT = \*EXIT-ROUTINE(...)**

The output destination is a user-owned subroutine which is loaded dynamically by BINDER with DBL macro BIND (see the “BLSSERV Dynamic Binder Loader / Starter” manual [1]). The following register conventions must be observed:

- |                         |                                                                                                                                                                                                                              |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Register 1:<br>(input)  | contains the address of a field with 2 elements:<br>element 1: address of the file name, which may be up to 54 characters in length<br>element 2: address of the parameter list, which may be up to 255 characters in length |
| Register 1:<br>(output) | contains the standard return code from the BIND macro                                                                                                                                                                        |
| Register 14:            | contains the return address                                                                                                                                                                                                  |

**ROUTINE-NAME = <text 1..32>**

Specifies the name of the user-owned subroutine.

**LIBRARY =**

Specifies the library which contains the user-owned subroutine.

**LIBRARY = \*BLSLIB-LINK**

The libraries with the file link name BLSLIBnn ( $00 \leq nn \leq 99$ ) are searched for the subroutine. The libraries are searched in *ascending* order of the values “nn” in the file link names.

**LIBRARY = <filename 1..54 without-gen-vers>**

Explicit specification of the library name.

**LIBRARY = \*LINK(...)**

The library is specified via its file link name.

**LINK-NAME = <structured-name 1..8>**

The file link name of the library.

**FILE-NAME = \*STD / <filename 1..54 without-gen-vers>**

Specifies the name (or link name) of the ISAM file into which the list is to be output. By default, the file name `BNDMAP.date.time.<tsn>` is used (see the `OUTPUT=BY-SHOW-FILE(...)` operand for the format of this name). The ISAM keys contained in the file are described in the appendix (see [page 369f](#)).

**LINE-SIZE = 136 / <integer 132..255>**

Specifies the number of characters per line.

**USER-PARAMETERS = \*NONE / <c-string 1..255 with-low> / <text 1..255>**

Specifies the parameters which are to be passed with the macro call `BIND`.



## SHOW-SYMBOL-INFORMATION

This statement outputs information about symbols on SYSOUT. The following information may be output:

- the logical position of the symbols in the LLM (i.e. the name of the module in which they are located)
- the attributes of the symbols
- the address of the slice in which the symbols are located.

This information can be requested for:

- all visible program definitions
- the initialized COMMONs
- the resolved external references
- all visible program definitions which have duplicates
- the unresolved external references.

### SHOW-SYMBOL-INFORMATION

```

SYMBOL-NAME = *ALL / list-poss(40): <c-string 1..255 with-low> / <text 1..32>
,INFORMATION = *LOGICAL-POSITION / *ALL /
                list-poss(3): *LOGICAL-POSITION / *ATTRIBUTES / *ADDRESS
,SELECT = *ALL / *COMMON-PROMOTION / *EXTERN-RESOLUTION / *DUPLICATE-LIST /
          *UNRESOLVED-LIST(...)
          *UNRESOLVED-LIST(...)
          | REFERENCE-TYPE = *ALL / list-poss(4): *EXTRN / *VCON / *WXTRN / *XDSECT-R

```

#### **SYMBOL-NAME =**

Specifies the names of the symbols for which the information is to be displayed.

#### **SYMBOL-NAME = \*ALL**

Information is displayed for all symbols.

#### **SYMBOL-NAME = list-poss(40): <c-string 1..255 with-low> / <text 1..32>**

Information is displayed only for symbols with the specified name.

Wildcards may be specified.

Note: BINDER checks special data types <symbol> and <symbol-with-wild> (see [page 199](#)).

**INFORMATION =**

Specifies which information about the symbols is to be displayed.

**INFORMATION = \*LOGICAL-POSITION**

The names of the modules in which the symbols are located are displayed.

**INFORMATION = \*ALL**

All information is displayed.

**INFORMATION = list-poss(3): \*LOGICAL-POSITION / \*ATTRIBUTES / \*ADDRESS**

Specifies the list of the requested information.

**INFORMATION = \*ATTRIBUTES**

The address, the length and the attributes of program definitions are displayed.

**INFORMATION = \*ADDRESS**

The names of the slices in which the symbols are located are displayed.

**SELECT =**

Selects information about symbol handling.

**SELECT = \*ALL**

Information about all visible program definitions is displayed.

**SELECT = \*COMMON-PROMOTION**

The initialized COMMONs and the CSECTs by which they were initialized are displayed.

**SELECT = \*EXTERN-RESOLUTION**

The resolved external references and the symbols with which they were resolved are displayed.

**SELECT = \*DUPLICATE-LIST**

Information about program definitions with duplicates is displayed.

**SELECT = \*UNRESOLVED-LIST(...)**

Unresolved external references and their logical positions (i.e. the names of the modules in which the unresolved external references are located) are displayed.

**REFERENCE-TYPE = \*ALL / list-poss(3): \*EXTRN / \*VCON / \*WXTRN**

Specifies which types of references are to be included in the displayed information.

## START-LLM-CREATION

This statement creates a new LLM in the work area, thereby deleting the previous work area contents. The following attributes can be defined for the LLM:

- internal name (INTERNAL-NAME)
- internal version (INTERNAL-VERSION)
- physical structure (SLICE-DEFINITION)
- copyright information (COPYRIGHT)
- use of logical structure information and LSD information (INCLUSION-DEFAULTS).

BINDER then includes in this current LLM those modules specified by means of INCLUDE-MODULES statements.

The LLM created is saved as a type L element in a program library by means of the SAVE-LLM statement.

## START-LLM-CREATION

```

INTERNAL-NAME = <c-string 1..32 with-low> / <text 1..32>
,INTERNAL-VERSION = *UNDEFINED / <composed-name 1..24> / <c-string 1..24>
,SLICE-DEFINITION = SINGLE / *BY-ATTRIBUTES(...) / *BY-USER(...)
  *BY-ATTRIBUTES(...)
    | READ-ONLY = *NO / *YES
    | RESIDENT = *NO / *YES
    | PUBLIC = NO / *YES(...)
    | *YES(...)
    | | SUBSYSTEM-ENTRIES = *NONE / list-poss(40): <c-string 1..32 with-low> / <text 1..32>
    | RESIDENCY-MODE = *NO / *YES
  *BY-USER(...)
    | AUTOMATIC-CONTROL = *YES (...) / *NO
    | *YES(...)
    | | RELOAD-SLICE = *YES / *NO
    | EXCLUSIVE-SLICE-CALL = *NO / *YES
,COPYRIGHT = *PARAMETERS (...) / *NONE
  *PARAMETERS(...)
    | NAME = *SYSTEM-DEFAULT / <c-string 1..64 with-low>
    | YEAR = *CURRENT / <integer 1900..2100>
    | PATH-NAME = *NONE / <c-string 1..255 with-low> / <text 1..255>
    | ENTRY = *NONE / <c-string 1..32 with-low> / <text 1..32>
,INCLUSION-DEFAULTS = *PARAMETERS (...)
  *PARAMETERS(...)
    | LOGICAL-STRUCTURE = *WHOLE-LLM / *OBJECT-MODULES
    | TEST-SUPPORT = *NO / *YES

```

**INTERNAL-NAME = <structured-name 1..32>**

Defines the internal name of the LLM created. The internal name forms the root in the logical structure of the LLM (see [page 16ff](#)). The internal name is entered as the element name on saving the LLM in the program library if corresponding values are set for the ELEMENT operand in the SAVE-LLM statement (see SAVE-LLM statement).

**INTERNAL-VERSION =**

Defines the internal version of the LLM created. The internal version is used as element version on saving the LLM in the program library if corresponding values are set for the VERSION operand in the SAVE-LLM statement (see SAVE-LLM statement on [page 285](#)).

**INTERNAL-VERSION = \*UNDEFINED**

When the LLM is saved with the SAVE-LLM statement, the default value for the highest version for program libraries is assumed (see the "LMS" manual [3]).

**INTERNAL-VERSION = <composed-name 1..24> / <c-string 1..24>**

Internal version of the LLM.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**SLICE-DEFINITION =**

Defines the physical structure of the LLM.

**SLICE-DEFINITION = \*SINGLE**

The LLM consists of a single slice.

**SLICE-DEFINITION = \*BY-ATTRIBUTES(...)**

The LLM consists of slices formed by combination of the attributes of CSECTs (see [page 16ff](#)). If the BY-ATTRIBUTES operand is specified and if all suboperands are set to NO, then SINGLE is assumed. The operand values for READ-ONLY, RESIDENT, PUBLIC and RESIDENCY-MODE simply result in combination to form slices. They do not affect the individual CSECTs. Up to 16 different slices can be formed by combination of attributes.

**READ-ONLY = \*NO / \*YES**

Specifies whether the READ-ONLY attribute is to be taken into consideration when forming slices.

When YES is specified, BINDER forms separate slices for CSECTs with differing READ-ONLY attributes.

**RESIDENT = \*NO / \*YES**

Specifies whether the RESIDENT attribute is to be taken into consideration when forming slices.

When YES is specified, BINDER forms separate slices for CSECTs with differing RESIDENT attributes.

**PUBLIC =**

Specifies whether the PUBLIC attribute is to be taken into consideration when forming slices.

**PUBLIC = \*NO**

The attribute PUBLIC is not taken into consideration when forming slices.

**PUBLIC = \*YES(...)**

BINDER generates separate slices for CSECTs with differing PUBLIC attributes.

**SUBSYSTEM-ENTRIES = \*NONE / list-poss(40): <text 1..32>**

Specifies the symbols (CSECTs or ENTRYs) of the PUBLIC slice which may be used for resolving external references if the PUBLIC slice is loaded as a dynamic subsystem (see the “Introductory Guide to Systems Support” [9]).

**SUBSYSTEM-ENTRIES = \*NONE**

No symbols from this subsystem (of the PUBLIC slice) are used for resolving external references.

**SUBSYSTEM-ENTRIES = <text 1..32>**

The name of the CSECT or the ENTRY in the PUBLIC slice loaded as a subsystem which may be used for resolving external references.

Note: BINDER checks special data type <symbol> (see [page 199](#)).

**RESIDENCY-MODE = \*NO / \*YES**

Specifies whether the RMODE attribute is to be taken into consideration when forming slices.

When YES is specified, BINDER forms separate slices for CSECTs with differing RMODE attributes.

**SLICE-DEFINITION = \*BY-USER(...)**

The physical structure of the LLM is defined by the user by means of SET-USER-SLICE-POSITION statements (user-defined slices). Overlays can be defined here.

**AUTOMATIC-CONTROL = \*YES(...) / \*NO**

This is relevant only for overlays.

Specifies whether an Overlay Control Module (OCM) is linked into the created LLM in order to control automatic dynamic loading of the overlays.

**AUTOMATIC-CONTROL = \*YES(...)**

An OCM is linked into the created LLM to control automatic dynamic loading of overlays.

**RELOAD-SLICE=**

Specifies whether a loaded slice must be reloaded overwriting the previous slice in main memory.

**RELOAD-SLICE= \*YES**

An already loaded slice is reloaded overwriting the previous slice in main memory.

**RELOAD-SLICE= \*NO**

An already loaded slice is kept unchanged in main memory.

**EXCLUSIVE-SLICE-CALL =**

Of significance only for overlays; specifies whether external references between exclusive slices are to be resolved.

**EXCLUSIVE-SLICE-CALL = \*NO**

Specifies that BINDER only reports external references and does not resolve them if it detects references between exclusive slices.

**EXCLUSIVE-SLICE-CALL = \*YES**

Causes BINDER to resolve external references between exclusive slices i.e. the user must accept any errors that may occur.

**COPYRIGHT =**

Defines the copyright information that is entered in the LLM created. The copyright information consists of text and the year number.

**COPYRIGHT = \*PARAMETERS(...)****NAME =**

Text for the copyright information.

**NAME = \*SYSTEM-DEFAULT**

The value of the class 2 system parameter BLSCOPYN is to be taken over. This value is defined at system installation time (see the "Introductory Guide to Systems Support" [9]).

**NAME = <c-string 1..64>**

New text for the copyright information. If the text comprises blanks, no copyright information is entered.

**YEAR =**

Year number for copyright information.

**YEAR = \*CURRENT**

Current year number.

**YEAR = <integer 1900..2100>**

Explicit specification of the year number.

**COPYRIGHT = \*NONE**

No copyright information is entered.

**INCLUSION-DEFAULTS =**

Defines the use of the logical structure information and LSD information. This is the default value that is used in the INCLUDE-MODULES, REPLACE-MODULES and RESOLVE-BY-AUTOLINK statements of the same edit run if no specific values are specified in these statements. Logical structure information and LSD information are not included during saving of the LLM unless this is required both in the SAVE-LLM statement and in preceding INCLUDE-MODULES, REPLACE-MODULES or RESOLVE-BY-AUTOLINK statements.

**INCLUSION-DEFAULTS = \*PARAMETERS(...)****LOGICAL-STRUCTURE =**

Specifies whether the logical structure information is taken over from the modules into the current LLM when including or replacing modules.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

The logical structure information is not taken over. A structure comprising only object modules (OMs) is established in the current LLM.

**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is taken over into the current LLM when including or replacing modules.



Information on the existence of the list for symbolic debugging is provided in the "T&D" column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.



## START-LLM-UPDATE

This statement updates an LLM that is saved as a type L element in a program library. The START-LLM-UPDATE statement reads the LLM from the program library into the BINDER work area. After it is read in, the LLM becomes the *current LLM*, i.e. it has the same status as it had before being saved in the program library. The current LLM can then be processed in the work area.

Processing of the current LLM is terminated without implicit saving of the LLM. It is saved again as a type L element in a program library by means of the SAVE-LLM statement. If the new element retains the same element name and the same element version and `OVERWRITE=YES` is specified, the previous element in the program library will be overwritten. The user should remember that START-LLM-UPDATE (and also INCLUDE-/REPLACE-MODULES) opens the input source in read-only mode and that other tasks may also have read-only access to the same source. In this case, the LLM cannot be stored with the same element name and the same version as the original LLM.

**START-LLM-UPDATE**

**MODULE-CONTAINER** = **\*LIBRARY-ELEMENT** (...) / **\*FILE**(...)

**\*LIBRARY-ELEMENT**(...)

**LIBRARY** = **\*CURRENT** / <filename 1..54 without-gen-vers> / **\*LINK**(...)

**\*LINK**(...)

**LINK-NAME** =

                <structured-name 1..8> / <filename 1..8 without-gen>

**,ELEMENT** = <composed-name 1..64>(…) / <c-string 1..64>(…)

        <composed-name>(…)

**VERSION** = **\*HIGHEST-EXISTING** / <composed-name 1..24> / <c-string 1..24>

        <c-string>(…)

**VERSION** = **\*HIGHEST-EXISTING** / <composed-name 1..24> / <c-string 1..24>

**\*FILE**(...)

**FILE-NAME** =

        <filename 1..54 without-gen-vers> / **\*LINK**(...)

**\*LINK**(...)

**LINK-NAME** =

            <structured-name 1..8> / <filename 1..8 without-gen>

**,INCLUSION-DEFAULTS** = **\*PARAMETERS** (...)

**\*PARAMETERS**(...)

**LOGICAL-STRUCTURE** = **\*UNCHANGED** / **\*WHOLE-LLM** / **\*OBJECT-MODULES**

**,TEST-SUPPORT** = **\*UNCHANGED** / **\*NO** / **\*YES**

**MODULE-CONTAINER =**

Defines where the LLM is stored.

**MODULE-CONTAINER = \*LIBRARY-ELEMENT(...)**

The LLM is stored in a program library.

**LIBRARY =**

Specifies the program library containing as an element the LLM that is to be updated.

**LIBRARY = \*CURRENT**

The program library specified in the most recent preceding START-LLM-UPDATE or SAVE-LLM statement is to be used. The scope of the operand / relates to one edit run.

**LIBRARY = <filename 1..54 without-gen-vers>**

File name of the program library that contains the LLM as an element.

**LIBRARY = \*LINK(...)**

Denotes a library by means of the file link name

**LINK-NAME = <structured-name 1..8> / <filename 1..8 without-gen>**

File link name of the program library.

**ELEMENT =**

Element name and element version of the LLM in the program library.

**ELEMENT = <composed-name 1..64>(…)**

Element name of the LLM.

Note: BINDER checks special data type <element-name> (see [page 199](#)).

**VERSION =**

Element version of the LLM.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**VERSION = \*HIGHEST-EXISTING**

The default value for the highest version in the program library is assumed (see the “LMS” manual [3]).

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

**ELEMENT = <c-string 1..64>(…)**

Element name of the LLM.

Note: BINDER checks special data type <element-name> (see [page 199](#)).

**VERSION =**

Version of the LLM.

**VERSION = \*HIGHEST-EXISTING**

The default value for the version identifier in the program library is assumed (see the “LMS” manual [3]).

**VERSION = <composed-name 1..24> / <c-string 1..24>**

Explicit specification of the element version.

Note: BINDER checks special data type <element-version> (see [page 199](#)).

**MODULE-CONTAINER = \*FILE(...)**

The LLM is stored in a PAM file.

**FILE-NAME =**

Specifies the file in which the PAM-LLM is stored.

**FILE-NAME = <filename 1..54 without-gen-vers>**

Name of the PAM file in which the LLM is stored.

**FILE-NAME = \*LINK(...)****LINK-NAME = <structured-name 1..8>**

File link name of the PAM file in which the LLM is stored.

**INCLUSION-DEFAULTS =**

Defines the use of the logical structure information and LSD information. This is the default value that is used in the INCLUDE-MODULES, REPLACE-MODULES and RESOLVE-BY-AUTOLINK statements of the same edit run if no specific values are specified in these statements. Logical structure information and LSD information are only transferred during saving of the LLM if this is required both in the SAVE-LLM statement and in preceding INCLUDE-MODULES, REPLACE-MODULES or RESOLVE-BY-AUTOLINK statements.

**INCLUSION-DEFAULTS = \*PARAMETERS(...)****LOGICAL-STRUCTURE =**

Specifies whether the logical structure information is taken over from the modules into the current LLM when including or replacing modules.

**LOGICAL-STRUCTURE = \*UNCHANGED**

The value defined in the START-LLM-CREATION statement on creating the LLM is applicable.

**LOGICAL-STRUCTURE = \*WHOLE-LLM**

All the logical structure information is taken over into the current LLM.

**LOGICAL-STRUCTURE = \*OBJECT-MODULES**

The logical structure information is not taken over. A structure consisting only of object modules (OMs) is created in the current LLM.

**TEST-SUPPORT =**

Specifies whether the LSD information from the modules is taken over into the current LLM when including or replacing modules.



Information on the existence of the list for symbolic debugging is provided in the “T&D” column in the BINDER lists. The TEST-SUPPORT field in BINDER lists only shows the setting of this TEST-SUPPORT operand.

**TEST-SUPPORT = \*UNCHANGED**

The value defined in the START-LLM-CREATION statement on creating the LLM is applicable.

**TEST-SUPPORT = \*NO**

The LSD information is not taken over.

**TEST-SUPPORT = \*YES**

The LSD information is taken over.

## START-STATEMENT-RECORDING

This statement starts statement recording. All BINDER statements entered after this statement are recorded in the memory. The //STOP-STATEMENT-RECORDING statement terminates recording. Output of the recorded statements in BINDER lists is controlled by the STATEMENT-LIST operand of the //SHOW-MAP statement.

<b>START-STATEMENT-RECORDING</b>
<b>STATEMENT-FORM= *SHORT / *LONG</b> <b>,RECORDING-MODE= *NEW / *EXTEND</b>

### STATEMENT-FORM =

Controls recording of default values.

### STATEMENT-FORM = \*SHORT

Only those operands are recorded whose value deviates from the default.

### STATEMENT-FORM = \*LONG

All operands are recorded, even if their value matches the default.

### RECORDING-MODE= \*NEW / \*EXTEND

Controls handling of log entries for statements which were generated by a preceding //START-STATEMENT-RECORDING statement in the same BINDER run.

### RECORDING-MODE= \*NEW

Older log entries are deleted from the memory.

### RECORDING-MODE= \*EXTEND

Older log entries are retained.

### Notes

In the event of memory saturation, recording is terminated with the message BND2102.

If a BINDER statement which is to be recorded is terminated abnormally, the log entry concerned may be incomplete.

## STOP-STATEMENT-RECORDING

This statement terminates statement recording started with //START-STATEMENT-RECORDING. BINDER statements entered after this statement are no (longer) recorded in the memory.

Furthermore, this statement can be used to delete the entries for statements which have already been recorded from the memory.

<b>STOP-STATEMENT-RECORDING</b>
<b>DELETE-RECORDED-STMT= *NO / *YES</b>

### **DELETE-RECORDED-STMT =**

Controls the handling of existing log entries.

### **DELETE-RECORDED-STMT= \*NO**

The log entries are retained in the memory.

### **DELETE-RECORDED-STMT= \*YES**

The log entries are deleted from the memory.





---

## 8 Usage models for generating LLMs

This chapter describes the different usage models for a program or set of programs generated in LLM format.

The described usage models are oriented to the properties that the created program must possess. For example, the model for generating a stand-alone program differs from that of a module that must be dynamically linked and loaded.

If LLMs were not used, various program formats with different BLS properties would have to be generated:

- a load module (program file or type C library element in a program library) for a stand-alone program
- a link module (type R library element).

This difference is eliminated by using LLMs. It is therefore necessary to define different usage models

The following usage models for LLMs can be distinguished on the basis of their different usage potential and main operational properties (dynamic linking/loading, performance, partial replacement and reconfiguration):

1. Program
2. Module
3. Program library
4. Module library

## 8.1 Program

This model is used for generating stand-alone executable programs.

### Main properties

This model has the following main properties:

- optimum loading performance
- BLS metadata stored in a program must be reduced to a minimum; the object can therefore not be modified by BINDER (e.g. via the REPLACE-MODULES statement)

The LLM format also allows the following features:

- the symbol name length is not limited to 8 characters
- an LLM can be loaded (partially or fully) as shared code (e.g. as a DSSM subsystem)

Reducing the BLS metadata in a program is an essential requirement for avoiding unexpected external reference resolution by an existing LLM during program generation (or loading).

### Possible reconfigurations

The following reconfigurations are possible for this model:

- Installation of a completely new program version (as for a phase). The LLM container is not locked during execution.
- Dynamic unloading of the program (or its public part), which is loaded as a DSSM subsystem, with the STOP-SUBSYSTEM statement and loading of a new version with the START-SUBSYSTEM statement.
- Dynamic unloading of the program (or its public part), which is loaded as user shared code, with the DSHARE macro and loading of a new version with the ASHARE macro.

The following applies to the last two cases: if the new private program part is loaded before the public part is available, the public part is loaded into task-local memory (BLS executes a time stamp validation to avoid inconsistencies between the private and public parts).

### **Correction options**

The following correction options exist for a program:

- using an REP file for online correction loading if the external symbol dictionary contains at least one CSECT
- using static object correction via LMS (MODIFY-ELEMENT statement)

### **Methods of use**

This type of LLM can be used in the following ways:

- It can be called with the START-PROGRAM-EXECUTABLE (or START-<program>) command.
- It can be called with the BIND macro, but only via the main program entry point

### **Restrictions**

The following restrictions are placed on this kind of LLM:

- It cannot be statically linked into another program
- It cannot be referenced by external programs via specific ENTRYs, it can only be called as a whole

## 8.2 Module

The purpose of this model is to generate modules which can be inserted into a program or dynamically loaded (via the dynamic resolution of external references or by calling the BIND macro).

It is typically used during migration for replacing the object modules (OMs) generated by the compiler.

### Main properties

The main properties of this model are:

- the loading performance compared to programs is not optimal due to dynamic linking
- the loading performance is, however, drastically improved compared to the old OM format (a factor of 3 to 10 has been proven).
- BLS metadata in the program must be reduced to a minimum. Only the names referenced by external modules/programs may be contained or visible in the module. These properties are comparable to those of an object module.

The LLM format also offers the following features:

- the symbol name length is not limited to 8 characters
- an LLM can be loaded (partially or fully) as shared code (e.g. as a DSSM subsystem)
- an LLM can be updated (e.g. with the REPLACE-MODULES statement) as long as the required data exists in storage

### Possible reconfigurations

The following reconfigurations are possible for this model:

- Dynamic unloading of a module loaded into task local memory with the UNBIND macro and loading of a new module with the BIND macro. This option is also available for the OM format.
- Dynamic unloading of the module, which is loaded as a DSSM subsystem, with the STOP-SUBSYSTEM statement and loading of a new version with the START-SUBSYSTEM statement. This option is also available for the OM format.
- Dynamic unloading of the module, which is loaded as user shared code, with DSHARE and loading of a new version with ASHARE. This option is also available for the OM format.

The following applies to the last two cases: if the new private part is loaded before the public part is available, the public part is loaded into task-local memory (BLS executes a time stamp validation to avoid inconsistencies between the private and public parts).

### Correction options

The following correction options exist for a module:

- Static LLM modification via BINDER (e.g. REPLACE-MODULES statement).
- Using an REP file for online correction loading.
- Using static object correction via LMS (MODIFY-ELEMENT statement).

### Methods of use

This type of LLM can be used in the following ways:

- It can be statically linked into another program.
- It can be referenced by other programs via specific entry points, a BIND macro call or the BLS autolink function

### Restriction

If this type of LLM is to be called with the START-EXECUTABLE-PROGRAM (or START-<program>) command, it is generally necessary to assign alternative libraries (link name BLSLIBnn) for the BLS autolink function.

## 8.3 Program library

The purpose of this model is to collect a set of programs belonging to the same application into a single library. Each of these programs has the same properties as a single program.

The following aspects must be taken into account:

- In most cases, it is absolutely imperative that dynamic resolution of external references between different programs is avoided. The external symbol dictionary must therefore be reduced to the minimum required.
- If part of the program contains shared code, this can be loaded as an application-specific subsystem.

### Possible reconfigurations

The following reconfigurations are possible for this model:

- Exchanging the complete application:
  - Installation of a complete new version of the program library (as for a phase library).
  - Dynamic unloading of the programs (or their public parts), which are loaded as a DSSM subsystem, with the STOP-SUBSYSTEM statement and loading of a new version with the START-SUBSYSTEM statement.
  - Dynamic unloading of the programs (or their public parts), which are loaded as user shared code, with the DSHARE macro and loading of a new version with the ASHARE macro.

If the library is exchanged before the shared part is unloaded/loaded, execution of a program leads to both the public and private parts being loaded into task-local memory.

- Exchanging separate programs in the library:

If the public parts of the programs are loaded as a DSSM subsystem, the complete subsystem must be unloaded and then reloaded. Separate programs can be unloaded/reloaded if user shared code is employed (in this case, however, the tasks which execute the shared parts are not validated).

If the public parts are not reloaded, executing one of the new programs leads to both the private and public parts being loaded into task-local memory.

### Correction options

The following correction options exist for this model:

- Using an REP file for online correction loading if the external symbol dictionary contains at least one CSECT. A single REP file can be used for the complete library but, in this case, a NOREF file must be used to prevent the message 'Rep name error' being output.
- Using static object correction via LMS (MODIFY-ELEMENT statement).

### Methods of use

This type of LLM can be used in the following ways:

- It can be called with the START-EXECUTABLE-PROGRAM (or START-<program>) command.
- It can be called with the BIND macro, but only via the main entry point of the program.

### Restrictions

The following restrictions are placed on this kind of LLM:

- It cannot be statically linked into another program.
- It cannot be referenced by other programs via specific entry points, it can only be called as a whole

## 8.4 Module library

The purpose of this model is to collect a set of modules belonging to the same application into a library. Language module runtime libraries are a typical example. Each of these modules has the same properties as a single module.

If some of the modules contain shared code, they can be loaded as an application-specific subsystem.

### Possible reconfigurations

The following reconfigurations are possible for a module library:

- Exchanging the complete application:
  - Installation of a complete new version of the module library (as for a phase library).
  - Dynamic unloading of the modules (or their public parts), which are loaded as a DSSM subsystem, with the STOP-SUBSYSTEM statement and loading of a new version with the START-SUBSYSTEM statement.
  - Dynamic unloading of the modules (or their public parts), which are loaded as user shared code, with the DSHARE macro and loading of a new version with the ASHARE macro.

If the library is exchanged before the shared part is unloaded/loaded, execution of a module leads to both the public and private parts being loaded into task local memory.

- Exchanging separate modules in the library:

If the public parts of the modules are loaded as a DSSM subsystem, the complete subsystem must be unloaded and then reloaded. Separate modules can be unloaded/reloaded if user shared code is employed (in this case, however, the tasks which execute the shared parts are not validated). If the public parts are not reloaded, loading one of the new modules leads to both the private and public parts being loaded into task local memory.

### Correction options

The following correction options exist for this model:

- static LLM modification via BINDER (e.g. REPLACE-MODULES statement)
- using an REP file for online correction loading
- using static object correction via LMS (MODIFY-ELEMENT statement)



**Methods of use**

This type of LLM can be used in the following ways:

- It can be statically linked into another program
- It can be referenced by external programs via specific entry points, BIND call or the BLS autolink function

**Restriction**

If this type of LLM is to be called with the START-EXECUTABLE-PROGRAM (or START-<program>) command, it is generally necessary to assign alternative libraries (link name BLSLIBnn) for the BLS autolink function.

## 8.5 Generating the different program types

This section describes how LLMs have to be created in order to conform to the different usage models.

### 8.5.1 Generating a program

A program is a stand-alone executable LLM. It can reference other objects (via dynamic resolution of external references or BIND macro calls). It can, however, neither be referenced by external objects nor statically linked into other LLMs. It may also be loaded as shared code.

#### 8.5.1.1 Non-shareable program

The LLM may be a single slice or divided into multiple slices according to the attributes READ-ONLY/READ-WRITE and/or RMODE=ANY/RMODE=24. The LLM can be saved without the external symbol dictionary or the logical structure. The relocation information may be saved with it if required.

#### Link procedure

```
//START-LLM-CREATION INTERNAL-NAME=<internal-name>, -
//      INTERNAL-VERSION=<version>
//INCLUDE-MODULES ... _____ (1)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (2)
//SAVE-LLM LIBRARY=<library>,ELEMENT=<element>, -
//      SYMBOL-DICTIONARY=*NO, LOGICAL-STRUCTURE=*NONE,
//      RELOCATION-DATA=*YES/*NO _____ (3)
```

- (1) All modules of the program (OMs or LLMs) are explicitly included.
- (2) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (3) The LLM is saved without external symbol dictionary or logical structure. The relocation information may be saved with it if required.

## Program execution

The program can be executed with the command:

```
/START-EXECUTABLE-PROGRAM -
/      FROM-FILE=*LIBRARY-ELEMENT( -
/          LIBRARY=<library>, ELEMENT-OR-SYMBOL=<element>), -
/      DBL-PARAMETERS=(LOADING=(PROGRAM-MODE=*ANY, REP-FILE=<rep-filename>))
```

It is also possible (and recommended) to create a specific SDF START-<program> command and, for example, implement it via a BS2000 procedure. This allows the parameters required to start program execution (e.g. the REP file name) to be 'hidden'. It also provides independence from the file names required for loading the program.

### 8.5.1.2 Partially shareable program

A partially shareable program contains shareable and non-shareable code. A partially shareable LLM must be divided into multiple slices according to the private/public attributes. The compiler generates the CSECT attribute automatically (see 'Compiling' below). BINDER then collects all CSECTs with the same attribute into one slice (see 'Link procedure' below). A DSSM catalog must be generated (see 'DSSM declaration' below) in order to load the public slice as a DSSM subsystem. The model is based on LLM format 2.

## Compiling

The sources must be compiled with the following option to generate CSECTs with public or private attributes (example of a C compiler):

```
COMPILER-ACTION=*MODULE-GENERATION(SHAREABLE-CODE=*YES,
MODULE-FORMAT=*OM/*LLM)
```

If symbol names longer than 8 characters are used, MODULE-FORMAT=\*LLM must be specified.

**Link procedure**

```
//START-LLM-CREATION INTERNAL-NAME=<name>, -
//  INTERNAL-VERSION=<version>, -
//  SLICE-DEFINITION=*BY-ATTRIBUTES(PUBLIC=*YES -
//                                (SUB-ENTRIES=<symbolname>)) _____ (1)
//INCLUDE-MODULES ... _____ (2)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (3)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=*ALL,VISIBLE=*NO _____ (4)
[//RENAME-SYMBOL SYMBOL-NAME=<public-definition>, -
//  NEW-NAME=<symbolname>,SYMBOL-OCCURENCE=*PARAMETERS -
//  (FIRST-OCCURENCE=1,OCCURENCE-NUMBER=*ALL)] _____ (5)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=<symbolname>, -
//  VISIBLE=*YES _____ (6)
//SAVE-LLM LIBRARY=<library>,ELEMENT=<element>
```

- (1) Slices can be defined with the **START-LLM-CREATION** (or **MODIFY-LLM-ATTRIBUTES**) statement. For **SUB-ENTRIES**, the user must specify a program definition (**CSECT** or **ENTRY**) in the public slice which is used in the private slice. The specified name must uniquely identify the subsystem. If this is not possible, the symbol name must be modified (see 5 below).
- (2) All program modules (OMs or LLMs) are included explicitly.
- (3) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (4) All symbols are masked.
- (5) Optional: If none of the program definitions in the LLM can be used as a unique specification for **SUB-ENTRIES**, the user must rename a public definition used in the private slice to create a unique name.
- (6) The required symbol is made visible.

**DSSM declaration**

The following definitions must be made in the subsystem catalog (with the product SSCM) to load the LLM public slice:

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<subsystem-name>,-
//  LIBRARY=<libraryname>,LINK-ENTRY=<symbolname>,-
//  SUBSYSTEM-ENTRIES=(<symbolname>(CONNECTION-SCOPE=*PROGRAM)),-
//  MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended).

### Preloading the public slice

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, the public slice is loaded into user address space when the program is executed.

### Program execution

The program can be executed with the following command:

```
/START-EXECUTABLE-PROGRAM -
/   FROM-FILE=*LIBRARY-ELEMENT( -
/       LIBRARY=<library>,ELEMENT-OR-SYMBOL=<element>), -
/   DBL-PARAMETERS=(LOADING=(PROGRAM-MODE=*ANY,REP-FILE=<rep-filename>))
```

It is also possible (and recommended) to create a specific SDF START-<program> command and, for example, implement it via a BS2000 procedure. This allows the parameters required to start program execution (e.g. the REP file name) to be 'hidden'. It also provides independence from the file names required for loading the program. For a partially shareable program, BLS then loads the private slice into the user address space and connects it with the corresponding public slice (using the symbol names). In order to avoid inconsistencies, BLS uses a time stamp to check whether the two slices belong to the same LLM and if the check fails, BLS loads the public slice into the user address space.

#### 8.5.1.3 Totally shareable program

A totally shareable program only contains shareable code. The user can create a totally shareable LLM as a single-slice LLM. In this case, the external symbol dictionary may only contain a single symbol which must be used as a connectable entry and as the entry point for program loading. A DSSM catalog must be generated in order to load the LLM as a DSSM subsystem (see 'DSSM declaration' below).

**Link procedure**

```
//START-LLM-CREATION INTERNAL-NAME=<internal-name>, -
//                      INTERNAL-VERSION=<version>
//INCLUDE-MODULES ... _____ (1)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (2)
//MERGE-MODULES NAME=<internal-name>,ENTRY-LIST=<symbol> _____ (3)
//SAVE-LLM LIBRARY=<library>,ELEMENT=<element>, -
// LOGICAL-STRUCTURE=*WHOLE-LLM, SYMBOL-DICTIONARY=*YES, -
// RELOCATION-DATA=*YES
```

- (1) All program modules (OMs or LLMs) are included explicitly.
- (2) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (3) All symbols apart from <symbol> are removed from the external symbol dictionary. After the MERGE-MODULES statement, the LLM contains just one CSECT and its properties apply to the whole program (e.g. READ-ONLY, RMODE, AMODE,...). If this is not acceptable, different MERGE-MODULE statements can be issued (this statement can be used at the sub-LLM or OM level).

**DSSM declaration**

The following must be defined in the subsystem catalog (with the product SSCM) for loading the LLM:

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<subsystem-name>, -
// LIBRARY= <libraryname>, LINK-ENTRY=<symbol>, -
// SUBSYSTEM-ENTRIES=(<symbol>(CONNECTION-SCOPE=*PROGRAM)), -
// MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended).

**Preloading the program**

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, the public slice is loaded into user address space when the program is executed.

## Program execution

The program can be executed with the following command:

```
/START-EXECUTABLE-PROGRAM -
/      FROM-FILE=*LIBRARY-ELEMENT( -
/      LIBRARY=<library>, ELEMENT-OR-SYMBOL=<element>), -
/      DBL-PARAMETERS=(LOADING=(PROGRAM-MODE=*ANY, REP-FILE=<rep-filename>))
```

It is only meaningful to specify <library> if the program is not loaded as a DSSM subsystem.

It is also possible (and recommended) to create a specific SDF START-<program> command and, for example, implement it via a BS2000 procedure. This allows the parameters required to start program execution (e.g. the REP file name) to be 'hidden'. It also provides independence from the file names required for loading the program.

## 8.5.2 Generating a module

A module is an LLM that can be included in a program and loaded dynamically (via dynamic resolution of external references or by calling the BIND macro). The aim of the models described in this section is to hide or suppress all symbols in the external symbol dictionary that are not required for calling the module, thus improving the loading performance (CPU and I/O).

### 8.5.2.1 Non-shareable module

A non-shareable module is loaded into the user address space and not in shared code.

The merge function can be used to suppress all definitions in the external symbol dictionary that are not required and retain only the required definitions. This is specified in the MERGE-MODULES statement. The number of symbols that may remain in the external symbol dictionary when using this function is, however, limited to 40. This method is shown in variant 1 of the link procedure.

The merge function cannot be used if the LLM contains more than 40 external names. To prevent name conflicts and unexpected resolving of external references, the user must mask all symbols not required at the module interface. In this way, symbols which are used only internally are not visible (variant 2 of the link procedure).

**Link procedure (variant 1 for a maximum of 40 external names)**

```
//START-LLM-CREATION INTERNAL-NAME=<internal-name>, -
//                      INTERNAL-VERSION=<version>
//INCLUDE-MODULES ... _____ (1)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (2)
//MERGE-MODULES NAME=<internal-name>,ENTRY-LIST=<list> _____ (3)
//SAVE-LLM LIBRARY=...,ELEMENT=...,
```

- (1) All program modules (OMs or LLMs) are included explicitly.
- (2) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (3) All symbols in the external symbol dictionary which are not required at the module interface are suppressed. The symbols that are likely to be referenced by external entities and are therefore to remain in the external symbol dictionary can be specified with <list>.

After the MERGE-MODULES statement, the LLM contains just one CSECT and its properties apply to the whole program (e.g. READ-ONLY, RMODE, AMODE,...). If this is not acceptable, different MERGE-MODULE statements can be issued (this statement can be used at the sub-LLM or OM level).

**Link procedure (variant 2 for modules with more than 40 external names)**

```
//START-LLM-CREATION INTERNAL-NAME=<internal-name>, -
//                      INTERNAL-VERSION=<version>
//INCLUDE-MODULES ... _____ (1)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (2)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=*ALL,VISIBLE=*NO _____ (3)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=<list 1>,VISIBLE=*YES _____ (4)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=<list 2>,VISIBLE=*YES
//SAVE-LLM LIBRARY=...,ELEMENT=...,
```

- (1) All program modules (OMs or LLMs) are included explicitly.
- (2) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (3) All symbols are masked.
- (4) The required symbols are made visible. The symbols that are likely to be referenced by external entities and should remain visible are specified with <list 1>/<list 2>. More than two statements can be specified if necessary.



### 8.5.2.2 Partially shareable module

Partially shareable modules contain both shareable code and non-shareable code. The shareable code can be loaded in shared code. It can also be loaded dynamically by dynamically resolving references. A partially shareable LLM must be divided into several slices formed in accordance with the public/private attributes. The CSECT attribute is generated automatically by the compiler (see “Compiling” below). BINDER then automatically groups all the CSECTs with the same attribute into a slice (see “Link procedure” below). A DSSM catalog must be generated so that the public slice can be loaded as the DSSM subsystem (see “DSSM declaration” below). The model is based on LLM format 2.

#### Compiling

The sources must be compiled with the following option to generate CSECTs with public or private attributes (example of a C compiler):

```
COMPILER-ACTION=*MODULE-GENERATION(SHAREABLE-CODE=*YES,  
MODULE-FORMAT=*OM/*LLM)
```

If symbol names longer than 8 characters are used, MODULE-FORMAT=\*LLM must be specified.

**Link procedure**

```

//START-LLM-CREATION INTERNAL-NAME=<name>,
//  INTERNAL-VERSION=<version>,SLICE-DEFINITION=*BY-ATTRIBUTES -
//  (PUBLIC=*YES(SUB-ENTRIES=<symbolname>)) _____ (1)
//INCLUDE-MODULES ... _____ (2)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (3)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=*ALL,VISIBLE=*NO _____ (4)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=<list>,VISIBLE=*YES _____ (5)
[//RENAME-SYMBOL SYMBOL-NAME=<public-definition>, -
//  NEW-NAME=<symbolname>,SYMBOL-OCCURENCE=*PARAMETERS -
//  (FIRST-OCCURENCE=1,OCCURENCE-NUMBER=*ALL)] _____ (6)
//SAVE-LLM LIBRARY=...,ELEMENT=...,

```

- (1) Slices can be defined with the START-LLM-CREATION (or MODIFY-LLM-ATTRIBUTES) statement. For SUB-ENTRIES, the user must specify a program definition (CSECT or ENTRY) in the public slice which is used in the private slice. The specified name must uniquely identify the subsystem. If this is not possible, the symbol name must be modified (see (6)).
- (2) All modules (OMs or LLMs) are included explicitly.
- (3) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (4) All symbols are masked.
- (5) The required symbols are made visible. The symbols that are likely to be referenced by external entities and should remain visible are specified with <list>. The symbol name specified for SUB-ENTRIES must remain visible.
- (6) Optional: If none of the program definitions in the LLM can be used as a unique specification for SUB-ENTRIES, the user must rename a (visible) public definition used in the private slice to create a unique name.

## DSSM declaration

The following definitions must be made in the subsystem catalog (with the product SSCM) to load the LLM public slice:

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<subsystem-name>, -
//  LIBRARY=<bibliotheksname>,LINK-ENTRY=<symbolname>, -
//    SUBSYSTEM-ENTRIES=(<symbolname> -
//                          (CONNECTION-SCOPE=*PROGRAM)), -
//  MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended). If the public slice can be accessed by external entities via names other than <symbolname>, these names must also be defined with the SUBSYSTEM-ENTRIES parameter:

```
SUBSYSTEM-ENTRIES=(<symbolname>(CONNECTION-SCOPE=*PROGRAM),
...<symbol 1>(CONNECTION-SCOPE=*PROGRAM),
...<symbol n>(CONNECTION-SCOPE=*PROGRAM))
```

## Preloading the public slice

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, the public slice is loaded into user address space when the program is executed.

### 8.5.2.3 Totally shareable module

Totally shareable modules contain only shareable code and can therefore be loaded in full in shared code. LLMs of this kind can also be loaded dynamically by dynamically resolving references. To generate a totally shareable LLM, you can use the models for non-shareable modules. (The variant with more than 40 external names is then defined.) A DSSM catalog must be generated so that the LLM can be loaded as the DSSM subsystem (see “DSSM declaration” below).

### Link procedure

```
//START-LLM-CREATION INTERNAL-NAME=<internal-name>, -
//                      INTERNAL-VERSION=<version>
//INCLUDE-MODULES ... _____ (1)
//INCLUDE-MODULES ...
//RESOLVE-BY-AUTOLINK ... _____ (2)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=*ALL,VISIBLE=*NO _____ (3)
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=<list>,VISIBLE=*YES _____ (4)
//SAVE-LLM LIBRARY=...,ELEMENT=...,
```

- (1) All modules (OMs or LLMs) are included explicitly.
- (2) The autolink function is called for the specified libraries (e.g. runtime libraries).
- (3) All symbols are masked.
- (4) The required symbols are made visible. The symbols that are likely to be referenced by external entities and should remain visible are specified with <list> (<list>=<symbol 1>,...,<symbol n>).

### DSSM declaration

The following definitions must be made in the subsystem catalog (with the product SSCM) to load the LLM public slice:

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<subsystem-name>, -
// LIBRARY=<libraryname>,LINK-ENTRY=<symbol 1>, -
// SUBSYSTEM-ENTRIES=(<symbol 1>(CONNECTION-SCOPE=*PROGRAM)), -
//                      ...<symbol n>(CONNECTION-SCOPE=*PROGRAM), -
// MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended).

### Preloading the module

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, the public slice is loaded into user address space when the program is executed.

### 8.5.3 Generating a program library

A program library is a set of programs that are loaded with START-EXECUTABLE-PROGRAM. It is used to define a product made up of a number of programs and may contain totally shareable, partially shareable and non-shareable programs. If the library contains shareable programs, these should be loaded as a DSSM subsystem. Since they are called with START-EXECUTABLE-PROGRAM, they must be generated together as stand-alone programs. The models described in this section are based on LLM format 2.

#### Compiling and linking

The various programs must be compiled and linked as described in the sections “[Non-shareable program](#)” on page 346, “[Partially shareable program](#)” on page 347 and “[Totally shareable program](#)” on page 349.

#### DSSM declaration

The following definitions must be made in the subsystem catalog (with the product SSCM) to load the program library (public slices of all partially and fully shareable LLMs):

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<symbolname>, -
//  LIBRARY=<programname>,LINK-ENTRY=<symbolname 1>, -
//  SUBSYSTEM-ENTRIES=(<symbolname 1> -
//                      (CONNECTION-SCOPE=*PROGRAM), -
//                      ...<symbolname n>(CONNECTION-SCOPE=*PROGRAM), -
//  MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

<symbolname 1> ... <symbolname n> are the names which were specified in the following operand when the individual programs were linked:

```
SLICE-DEFINITION=*BY-ATTRIBUTES(PUBLIC=*YES(SUB-ENTRIES=<symbolname>)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended).

#### Preloading the public slices

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, all shareable parts are loaded into user address space.

## Execution

The following command can be used for executing a specific program from the program library:

```
/START-EXECUTABLE-PROGRAM -  
/      FROM-FILE=*LIBRARY-ELEMENT( -  
/      LIBRARY=<library>, ELEMENT-OR-SYMBOL=<element>), -  
/      DBL-PARAMETERS=(LOADING=(PROGRAM-MODE=*ANY, REP-FILE=<rep-filename>))
```

It is also possible (and recommended) to create a specific SDF START-<program> command and, for example, implement it via a BS2000 procedure. This allows the parameters required to start program execution (e.g. the REP file name) to be 'hidden'. It also provides independence from the file names required for loading the program. For a partially shareable program, BLS then loads the private slice into the user address space and connects it with the corresponding public slice (using the symbol names). In order to avoid inconsistencies, BLS uses a time stamp to check whether the two slices belong to the same LLM. If the check fails, BLS loads the public slice into the user address space.

## 8.5.4 Generating a module library

A module library is a set of modules that can be statically linked into an application or dynamically loaded. Its purpose is to define products made up of multiple modules and it can contain fully, partially and non-shareable modules. If the library contains shareable modules, these should be loaded together as a DSSM subsystem. The models described in this section are based on LLM format 2.

### Compiling and linking

The various modules must be compiled and linked as described in the sections [“Non-shareable module” on page 351](#), [“Partially shareable module” on page 353](#) and [“Totally shareable module” on page 355](#).

### DSSM declaration

The following definitions must be made in the subsystem catalog (with the product SSCM) to load the module library (public slices of all partially and fully shareable LLMs):

```
//SET-SUBSYSTEM-ATTRIBUTES SUBSYSTEM-NAME=<subsystem-name>, -
//  LIBRARY=<programname>,LINK-ENTRY=<symbolname 1>, -
//  SUBSYSTEM-ENTRIES=(<symbolname 1>(CONNECTION-SCOPE=*PROGRAM), -
//                      ...<symbolname k>(CONNECTION-SCOPE=*PROGRAM), -
//  MEMORY-CLASS=*SYSTEM-GLOBAL(SUBSYSTEM-ACCESS=*LOW/*HIGH)
```

<symbolname 1> ... <symbolname n> are the names which were specified in the following operand when the individual modules were linked:

```
SLICE-DEFINITION=*BY-ATTRIBUTES(PUBLIC=*YES(SUB-ENTRIES=<symbolname>)
```

The subsystem access must be defined dependent on the program addressing mode (HIGH is recommended).

### Preloading the public slices

The subsystem can be loaded with the START-SUBSYSTEM command. It is also possible to specify automatic subsystem loading with the parameter CREATION-TIME=\*AFTER-SYSTEM-READY in the SSCM statement SET-SUBSYSTEM-ATTRIBUTES. If the subsystem is not loaded, all shareable parts are loaded into user address space.

## 8.6 More information about the usage models

### 8.6.1 LLM format 1

All models in this document refer to LLM format 2. LLM format 1 differs from LLM format 2 mainly in the following points:

With LLM format 1, BLS calls DSSM for each symbol in the public slice that is referenced by the private slice. All names in the DSSM catalog must therefore be defined as connectable entries, which considerably limits the use of such objects (the symbols corresponding to a shareable runtime module used in different LLMs were defined in all subsystems concerned).

With LLM format 2, the user can define a symbol (SUBSYSTEM-ENTRIES operand in START-LLM-CREATION) during the BINDER run that must be used for connecting the two slices at load time. This symbol must be in the public slice and referenced by the private slice. In this case, BLS only calls DSSM for this symbol and resolves all other external references automatically without calling DSSM. The user therefore only has to declare one symbol in the DSSM catalog. As of BINDER V1.3A, it is possible to generate LLM format 1 with the same property as format 2 in this respect.

### 8.6.2 Preloading public slices via the ASHARE interface

Only the DSSM functions for preloading shared code are described in this document.

A further option for loading shared code is “shared user code”. In this case, the shared code can be loaded via the BLS program interface ASHARE into the common memory pool in class 6 memory and a DSSM catalog is not required.

The memory pools used by the ASHARE interface for loading shared code are requested by the user program, and must therefore be managed by the user program itself (unlike subsystems, which are managed by DSSM).

You should note that a memory pool is released by the system if it is no longer linked to a task, i.e. after the last task has released the connection either explicitly with DISMP or implicitly when the program is terminated.



The following steps therefore need to be taken:

1. To keep the memory pool available throughout the entire run of an application, a program should be created to run in a separate task for memory pool management. This program will then start the application as follows:

- it creates the memory pool with ENAMP
- it loads the application code with ASHARE

The program then switches to a wait state until the application can be terminated. It then performs termination of the application:

- it unloads the code with DSHARE
- it releases the memory pool with DISMP

2. If the module is to be exchanged, it is recommended that you use the ILE mechanism of the DBL (see the manual “BLSSERV Dynamic Binder Loader/Starter” [1]). This enables you to exchange a module loaded as shared code without modifying the private part of the application. It is also recommended that you manage the connections to the PUBLIC modules if the exchanged modules are to be unloaded. You can do this with a “usage counter”, which is increased and decreased in the indirect linkage routine (by means of CS statements). DSHARE can then be executed as soon as the counter indicates that no further task which is to be unloaded is using the code.



---

## 9 Migration

This section highlights the differences between the old linkage editor/loader system (up to BS2000 V9.5) and the current Binder-Loader-Starter (BLS) system (as of BS2000 V10.0) and is intended to help the user make the transition.



For information on the linkage editor TSOSLNK and the static loader ELDE as well as information on migration from TSOSLNK to BINDER refer to the previous version of the manual (“BINDER V2.3”).

### 9.1 Old and current concepts

First, it is useful to compare and contrast concepts from the previous linkage editor/loader system and the new Binder-Loader-Starter system (see also [“Glossary” on page 373](#)).

#### 9.1.1 Old concepts

##### **Object module (OM)**

Loadable unit which is generated by translating a source program by means of a language processor.

##### **Object module library (OML)**

PAM file which contains object modules in the form of library elements. As of LMS V2.0A, object module libraries are no longer supported and should be replaced by program libraries.

##### **EAM object module file (OMF)**

Temporary system object module library in which object modules (OMs) or prelinked modules are stored by the language processor.

**Program library**

PAM file which is processed using the PLAM library access method. It contains library elements which are uniquely identifiable by the element type and element identifier.

**Segment**

Section of a program that can be loaded separately and executed independently of other program segments.

**DLL**

Dynamic linking loader of the previous linkage editor/loader system. It links object modules (OMs) and prelinked modules (GMs) into an executable program and loads this into computer memory.

## 9.1.2 Current concepts

### Link and load module (LLM)

Loadable unit that possesses a logical and a physical structure. It is generated by the linker BINDER and stored in a program library as a type L library element.

### Module

Generic term for an object module (OM) and a link and load module (LLM).

### Load unit

Contains all modules that are loaded with a *single* load call. Each load unit is located in a context.

### Context

A context can be:

- a set of load units
- a linking and loading environment
- an unloading and unlinking environment.

A context has a scope and an access privilege.

### Slice

Loadable unit comprising all the control sections (CSECTs) that are loaded contiguously. Slices form the physical structure of a link and load module (LLM).

### Binder-Loader-Starter (BLS)

Designation of the current linkage editor/loader system.

### BINDER

Linker (linkage editor) of the current Binder-Loader-Starter system. It links modules into a link and load module (LLM).

### DBL

Dynamic binder loader of the current Binder-Loader-Starter system. It links modules into a load unit and loads this into computer memory.

## 9.2 Features of the current Binder-Loader-Starter system

The current Binder-Loader-Starter (BLS) system is a significant departure from the concept underlying the old linkage editor/loader system. The key features of the current concept are summarized below.

### 9.2.1 Link and load module (LLM) and BINDER

- The format of the link and load module (LLM) permits optimization of loading compared with the object modules (OMs) and prelinked modules of the previous system.
- The logical structure of an LLM enables the user to structure his or her application. The user can remove or replace a node when creating or modifying an LLM, or include the node of an LLM in another LLM.
- An LLM that is saved in a program library can be modified. Modules can be replaced or additional modules included in the LLM.
- The user can specify that all CSECTs that have the same attributes or combination of attributes are to be combined into slices by BINDER. This significantly reduces the loading time and main memory requirements when the LLM is loaded.
- Symbols can be selected using wildcards, e.g. when modifying the visibility (masking/nonmasking) of symbols (MODIFY-SYMBOL-VISIBILITY).
- List for symbolic debugging (LSD) information can be included in an LLM for testing and diagnostic purposes. The user can select individual object modules (OMs) or sub-LLMs in which the LSD information is to be inserted.
- BINDER uses an SDF interface. The statements can be entered in dialog (interactive) mode or in batch mode. Each statement is processed *immediately* after input.
- At any time when creating an LLM the user can request lists to be output (using SHOW-MAP) containing information about the status of the current LLM. The user can then decide whether further modules are to be included in the current LLM or whether modules are to be removed. The lists are output to SYSLST or SYSOUT. BINDER implicitly uses the SDF SHOW-FILE command for this. During a BINDER run, the user has access not only to list output but also to other easy-to-use information functions.

## 9.2.2 Dynamic binder loader DBL

- The user can make use of contexts. This has the following advantages:
  - Multiple copies of the same program can be loaded into different contexts.
  - Parts of a comprehensive application can be loaded into different contexts. External references are resolved separately in each individual context. Each partial application in a context can therefore be loaded and started as an independent “sub-application”. In this way it is possible to load and start the individual modules of, for example, a runtime system in separate contexts.
  - Parts of an application belonging to a context can be unloaded with *one* call.
  - No name conflicts are caused by having identically named symbols in different contexts because each context has its own symbol table.
- When DBL is invoked, a list of information concerning the logical structure and contents of the loaded load unit can be requested.
- Up to 100 alternate libraries can be specified for the autolink function.
- The user can define according to individual requirements how name conflicts and unresolved external references are to be handled.
- When loading a load unit with the BIND macro, the user can specify that:
  - REP records from a REP file are to be applied to the modules of the load unit
  - libraries used by DBL are to remain open when processing of the DBL call has been completed. This can speed processing when DBL is called repeatedly with the same library.

### 9.2.3 DBL and BINDER

- One of the main features of the current Binder-Loader-Starter (BLS) system is the large degree of harmonization between BINDER and DBL. This means:
  - All LLM information that has been specified by BINDER when creating the LLM can be evaluated immediately by the new DBL.
  - External references are resolved by BINDER and DBL according to the same rules.
- Functions which the DLL could apply only to dynamically loaded object modules (e.g. modify options, information output, unloading) can be applied to the entire load unit by DBL.
- Both DBL and BINDER can process symbol names up to 32 characters long. These symbol names can be generated by BINDER by renaming the original symbol names in an LLM (RENAME-SYMBOLS).
- BINDER can generate LLMs with slices formed according to the PUBLIC or PRIVATE attribute of the CSECTs. The user can declare the PUBLIC slice as shareable by loading it in a common memory pool using the DBL macro ASHARE.



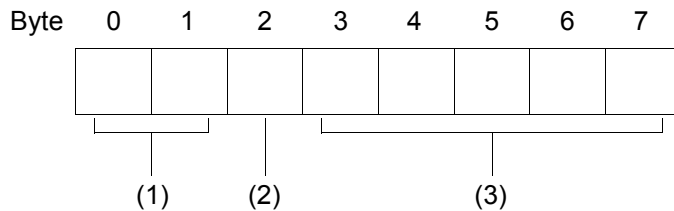
---

## 10 Appendix: Description of the ISAM keys

In the BINDER statements SHOW-MAP, MODIFY-MAP-DEFAULTS and SHOW-LIBRARY-ELEMENTS the user can select the medium to which the required information is to be output. If OUTPUT ≠ \*SYSLST is selected, BINDER outputs the information as an ISAM file. The keys of this ISAM file are eight bytes long and are described below.

### ISAM keys in BINDER lists (MAPs)

The ISAM key consists of three sections:



- (1) Bytes 0 through 1:  
indicate the section of the BINDER map:

05	HELP INFORMATION
10	GLOBAL INFORMATION
15	LOGICAL STRUCTURE
17	SCOPE PATH INFORMATION
20	PHYSICAL STRUCTURE if the LLM contains user-defined slices
25	PHYSICAL STRUCTURE if the LLM is sliced by attributes or only contains a single slice
30	PROGRAM MAP
31	COMMON LIST
35	UNRESOLVED REFERENCES
36	UNRESOLVED LONG NAMES
37	NOT REFERENCED SYMBOLS
40	SORTED SYMBOLS
45	PSEUDO REGISTERS
50	UNUSED MODULES
55	DUPLICATE SYMBOLS
60	INPUT INFORMATION
61	LINKNAME CONVERSION

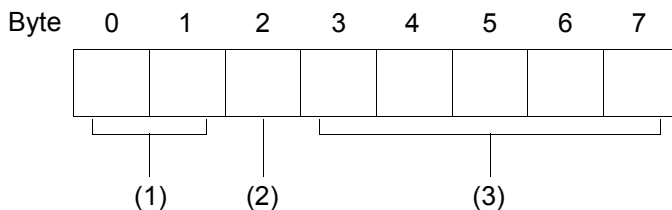
- (2) Byte 2:  
indicates the record type in the section of the BINDER map:

Decimal value	Meaning
0	header record
1	information record
9	end of section

- (3) Bytes 3 through 7:  
sequential number of the record in the relevant record type; decimal; starts at 0 for a new record type

## ISAM keys in the SHOW-LIBRARY-ELEMENTS lists

The ISAM key consists of three sections:



- (1) Bytes 0 through 1:  
indicate the type of list output for SHOW-LIBRARY-ELEMENTS:

Decimal value	Meaning
90	DUPLICATE SYMBOLS
50	SYMBOLS IN LIBRARY
00	LIBRARY CONTENT

- (2) Byte 2:  
indicates the record type in the selected list:

Decimal value	Meaning
0	header record
1	information record
9	end of section

- (3) Bytes 3 through 7:  
sequential number of the record in the relevant record type; decimal; starts at 0 for a new record type



---

# Glossary

This glossary contains definitions of key terms used in the description of the Binder-Loader-Starter (BLS) system. Cross-references are indicated by typeface *italic* of the associated term.

## **access privilege for a context**

Defines which users may access a *context*. The context can be privileged or nonprivileged.

## **addressing mode**

### **AMODE**

*Attribute of a control section (CSECT)*. Hardware addressing mode which a *program* or *load unit* expects at runtime. It can be set to:

- 24-bit addressing (AMODE=24)
- 31-bit addressing (AMODE=31)
- 24- and 31-bit addressing (AMODE=ANY).

### **AMODE**

*Addressing mode*

## **attribute**

Property which can be assigned to a *control section (CSECT)* at assembly or compilation time. A CSECT can have the following attributes:

- read access (READ-ONLY)
- memory-resident (RESIDENT)
- shareable (PUBLIC)
- *residence mode (RMODE)*
- alignment (ALIGNMENT)
- *addressing mode (AMODE)*.

## **autolink**

Automatic search and insert mechanism for including *modules*.

## **BINDER run**

Sequence of BINDER statements which begins after the load call for BINDER and ends with the END statement.

### **COMMON**

*Common section*

### **common memory pool**

A memory area in class 6 memory (user memory) which may be accessed by several users.

### **common section**

COMMON

Data area which can be shared by a number of *control sections (CSECTs)* for communication.

### **context**

A context can be:

- a set of objects with a *logical structure*
- an environment for linking and loading
- an environment for unloading and unlinking.

A context has a *scope* and an *access privilege*.

### **control section**

CSECT

Program section which can be loaded independently of other program sections. A control section can have certain *attributes*.

### **CSECT**

*Control section*

### **current LLM**

Newly created or modified *link and load module (LLM)* which is constructed in the BINDER work area.

### **current slice**

Slice into which *modules* are inserted or in which modules are replaced if nothing is defined concerning their position in the *physical structure of the LLM*. Applies to *user-defined slices* only.

### **EAM object module file (OMF)**

Temporary system object module library in which *object modules (OMs)* produced by a compiler are stored.

**edit run**

Comprises a sequence of BINDER statements which begins with the START-LLM-CREATION or START-LLM-UPDATE statement and ends with the next START-LLM-CREATION or START-LLM-UPDATE statement or with the END statement.

**element identifier**

Designates a library element in a *program library*; it is composed of the *element name* and *element version*.

**element name**

Name of a library element in a *program library* or *object module library*. It is referred to by the BINDER statements and the DBL commands.

**element type**

Type of a library element in a *program library*.

The following element types apply to program libraries:

- type C *program (load module)*
- type L *link and load module (LLM)*
- type R *object module (OM)*

**element version**

Version designation of a library element in a *program library*. It is referred to by the BINDER statements and the DBL commands.

**ENTRY**

*Entry point*

**entry point****ENTRY**

Symbolic link address which is defined in one *module* but can also be used by another module.

**ESD**

*External symbol dictionary*

**ESV**

*External symbols vector*

**external dummy section****XDSEC**

Program section for which there is no image in the *text information* of a module. An external dummy section can be a *reference (XDSEC-R)* or a *program definition (XDSEC-D)*.

### **external reference**

EXTRN

Symbolic link address which is used in one module but defined in another; it is resolved unconditionally either explicitly or by *autolink*.

### **external symbol dictionary**

ESD

Contains all the *program definitions* and *references* in a *module*. It is required for resolving references. Object modules (OMs) contain the ESD in the form of ESD records. Link and load modules (LLMs) contain *ESV* records.

### **external symbols vector**

ESV

Contains the *ESD* in *link and load modules (LLMs)*

### **ILE**

indirect linkage entry

*Entry point (ENTRY)* which the caller forwards to an *ILE server* by means of an *IL routine*. An ILE has the following attributes:

- name
- address of the *IL routine*
- address of the *IL server*
- displacement of the *ILE server* address in the *IL routine*
- status of the *ILE server* (active or not active)
- control indicator (system-driven or user-driven)

### **ILE server**

*Module* containing program code in the same way as a subprogram but which can be branched to via an *ILE* and an *IL routine*.

### **IL routine**

indirect linkage routine

Routine which calls an *ILE server*. Users can also define their own IL routine if they do not want to use the one provided by the system.

### **indirect linkage**

Linkage mechanism in which an *external reference* is resolved by means of an *ILE* and an intermediate *IL routine*, rather than directly by means of a program definition.

### **indirect linkage entry**

*ILE*



**indirect linkage routine**

*IL routine*

**internal name**

Defined when a *link and load module (LLM)* is created and identifies the root in the *logical structure of the LLM*.

**link and load module**

LLM

Loadable unit with a *logical structure* and a *physical structure*. LLMs are generated by BINDER and stored in a *program library* as type L library elements (*element type*) or in a PAM file (*PAM-LLM*).

**list for symbolic debugging**

LSD

Test and diagnostic information which is held in a *module* and which is required by the debugging and diagnostic tools for debugging at source program level.

LLM

*Link and load module*

**load unit**

Contains all *modules* that are loaded with a *single* load call. Each load unit is situated in a *context*.

**local relocation dictionary**

LRLD

Information in a *module* which determines how addresses are to be adjusted relative to a common base address during linking and loading.

**logical structure information**

Information in a *link and load module (LLM)* which determines the *logical structure of the LLM*.

**logical structure of a context**

Hierarchical structure of a *context* as a set of objects. Objects are *control sections (CSECTs)*, *modules* and *load units*.

**logical structure of an LLM**

Defines the tree structure of a *link and load module (LLM)*. The root forms the *internal name*; the nodes form *sub-LLMs*; the leaves are *object modules (OMs)* and empty sub-LLMs.

**LRLD**

*Local relocation dictionary*

**LSD**

*List for symbolic debugging.*

**module**

Generic term for *object module (OM)* and *link and load module (LLM)*.

**object module****OM**

Loadable unit generated by translating a source program by means of a language processor routine (assembler, compiler).

**object module library****OML**

PAM file which contains *object modules* as library elements.

**OM**

*Object module*

**OML**

*Object module library*

**PAM-LLM**

LLM which is stored by BINDER in a PAM file.

**path name**

Name by which *sub-LLMs* are addressed in the *logical structure of an LLM*. It consists of a sequence of individual names separated from one another by a period.

**physical structure information**

Information in a *link and load module (LLM)* which defines the *physical structure of the LLM*.

**physical structure of an LLM**

Defines the *slices* from which a *link and load module (LLM)* is constructed. These may be:

- *single slices*
- *slices by attributes*
- *user-defined slices*.

**program definition**

Generic term for

- *control section (CSECT)*
- *entry point (ENTRY)*
- *COMMON section*
- *external dummy section (XDSEC-D)*

**program library**

PAM file which is processed using the library access method PLAM. A program library contains library elements which are uniquely identifiable by *element type* and *element identifier*.

**pseudo-register**

Main memory area which is used for intercommunication by different program sections.

**pseudo-RMODE**

*Residence mode (RMODE)* of a *module*. It is defined by BINDER or DBL on the basis of the residence mode of all the *CSECTs* in the module.

**reentrant program**

A program whose code is not modified during execution. This is a prerequisite for use of the program as *shared code*.

### reference

Generic term for

- *external reference (EXTRN)*
- *V-type constant*
- *weak external reference (WXTRN)*
- *external dummy section (XDSEC-R)*

### residence mode

#### RMODE

*Attribute of a control section (CSECT); it defines whether the CSECT will be loaded above and below 16 Mb (RMODE=ANY) or only below 16 Mb (RMODE=24).*

#### RMODE

*Residence mode*

### scope of a context

Defines the memory class in which a *context* is situated. The context can be in the user address space (USER) or system address space (SYSTEM).

### shared (SHARE) program

Module which has been declared shareable by the system administrator by means of the ADD-SHARED-PROGRAM command; it is loaded into class 4 memory.

### single slice

*Physical structure of a link and load module (LLM) in which the LLM consists of a single slice.*

### shared code

Code which may be used simultaneously by several tasks. It may be stored in class 4 memory or in a *common memory pool* of class 6 memory. In order to be used as shared code, the program must have been written as a *reentrant program*.

### slice

Loadable unit which combines all the *control sections (CSECTs)* that are to be loaded together. Slices form the *physical structure of a link and load module (LLM)*.

### slices by attributes

*Physical structure of a link and load module (LLM) in which the slices are formed according to attributes of control sections (CSECTs)*

**sub-LLM**

Substructure in the *logical structure of an LLM*; it consists of *object modules (OMs)* or other sub-LLMs and is addressed by means of the *path name*.

**symbol**

Generic term for *program definition* and *reference*; it is identified by a symbol name.

**text information**

Information in a *module* which consists of the code and the data.

**user-defined slices**

*Physical structure* of a link and load module (LLM) in which the *slices* are defined by the user by means of SET-USER-SLICE-POSITION statements. Overlay structures can be formed at the same time.

**V-type constant**

Address constant which is defined in one *module* but whose address is used in another module; it is resolved unconditionally, either explicitly or by *autolink*.

**weak external reference****WXTRN**

Has the same characteristics as an *external reference (EXTRN)*, but is only resolved conditionally. *Autolink* cannot be applied to WXTRNs.

**WXTRN**

*Weak external reference*

**XDSEC**

*External dummy section*



---

## Related publications

You will find the manuals on the internet at <http://manuals.ts.fujitsu.com>. You can order printed copies of those manuals which are displayed with an order number.

- [1] **BLSSERV**  
**Dynamic Binder Loader / Starter in BS2000**  
User Guide
- [2] **ASSEMBH** (BS2000)  
User Guide
- [3] **LMS** (BS2000)  
SDF Format  
User Guide
- [4] **BS2000 OSD/BC**  
**System Installation**  
User Guide
- [5] **BS2000 OSD/BC**  
**Commands**  
User Guide
- [6] **JV** (BS2000)  
**Job Variables**  
User Guide
- [7] **BS2000 OSD/BC**  
**Executive Macros**  
User Guide
- [8] **AID** (BS2000)  
Advanced Interactive Debugger  
**Core Manual**  
User Guide

- [9] **BS2000 OSD/BC**  
**Introductory Guide to Systems Support**  
User Guide
- [10] **BS2000 OSD/BC**  
**System Messages**  
User Guide
- [11] **Introductory Guide to XS Programming**  
**(for Assembler Programmers) (BS2000)**  
User's Guide
- [12] **BS2000**  
**Programmiersystem** (only available in German)  
Technische Beschreibung  
(Programming System, Technical Description)
- [13] **SDF (BS2000)**  
**Introductory Guide to the SDF Dialog Interface**  
User Guide
- [14] **SDF-A (BS2000)**  
User Guide
- [15] **BS2000 OSD/BC**  
**Introductory Guide to DMS**  
User Guide



---

# Index

## A

abbreviation facilities 190  
address relocation 104  
addressing mode 114, 253  
alias 191, 194  
ALIGNMENT 114, 253  
alignment (ALIGNMENT) 114, 253  
alphanum-name (data type) 195  
AMODE 114, 253  
asterisk preceding constant operand value 190  
attributes of a CSECT 18  
attributes of an LLM, modifying 43  
attributes of modules, modifying 66, 242  
attributes of symbols 114  
    modifying 114, 253  
    modifying, example 114  
attributes, sliced by common promotion 103  
autolink function 80, 85  
    of BINDER 277  
    rules 85

## B

backtracking 25  
batch mode 15, 81  
BEGIN-SUB-LLM-STATEMENTS 52, 57, 209, 262  
beginning of a sub-LLM 57  
bind 9  
BINDER functions 35  
BINDER list output, example 143  
BINDER macro 171  
BINDER run 15, 80, 134, 138, 140, 167  
    monitoring with job variables 169  
    program information 170  
    starting 167

    status indicator 169  
    terminating 167, 212  
    termination code 169

BINDER runs, simultaneous 140, 288, 329  
BINDER statements 35  
BINDER-MAPs, ISAM keys 142  
BLSCOPYN 237, 327  
BLSLDPXS 296  
BLSLIBnn 80, 215, 278, 306  
    in REPLACE-MODULES 270  
BLSSEC 10

## C

c-string (data type) 195  
calling BINDER 167  
cat (suffix for data type) 206  
cat-id (data type) 195  
changing  
    logical name of an LLM 66  
    path name of a module 66  
class 2 system parameter  
    BLSCOPYN 237, 327  
    BLSLDPXS 296  
command-rest (data type) 195  
COMMON 103, 105  
    in name conflicts 102  
    modifying attributes 114, 253  
    renaming 107, 263  
COMMON promotion 103  
compilers 9  
composed-name (data type) 195  
constructor (string) 204  
contents of an LLM 29

- control section (CSECT) 18, 105
  - modifying attributes 114, 253
  - renaming 107, 263
  - searching for 80
- controlling error processing 134, 229
- controlling, logging 133
- COPYRIGHT 37, 43, 237, 327
- copyright information 37, 43, 237, 327
- corr (suffix for data type) 206, 207
- creating an LLM 30, 37, 323
  - example 39
- CSECT 18, 105
  - handling by autolink 277
  - modifying attributes 114, 253
  - renaming 107, 263
  - searching for 80
- CSECT attributes 18
- current
  - program library 43
  - slice 69
  - sub-LLM 57, 209
- current LLM 37, 40, 43, 285
  - resetting 212
  - storing 329
- D**
- data type
  - alphanum-name 195
  - c-string 195
  - cat-id 195
  - command-rest 195
  - composed-name 195
  - date 195
  - device 195
  - filename 196
  - fixed 195
  - integer 197
  - name 197
  - posix-filename 197
  - posix-pathname 198
  - product-version 198
  - structured-name 198
  - text 198
  - time 198
  - vsn 199
  - x-string 199
  - x-text 199
- data types in SDF 191, 195
  - suffixes 192
- date (data type) 195
- DBL (dynamic binder loader) 10, 18
- default file link name
  - for BINDER lists 310, 318, 319
- default file name for BINDER lists 310
- device (data type) 195
- DUPLICATE-LIST 317
  - in BINDER lists 142
- dynamic binder loader (DBL) 18
- E**
- EAM object module file (OMF) 9
  - in INCLUDE-MODULES 48, 213
  - in REPLACE-MODULES 54, 268
  - input source for BINDER 140
- edit run 38, 237
- ELEMENT
  - in INCLUDE-MODULES 215
  - in REPLACE-MODULES 271
  - in SAVE-LLM 23, 288
  - in SHOW-LIBRARY-ELEMENT 307
  - in START-LLM-UPDATE 331
- element identifier, definition 23
- element name
  - definition 23
  - in INCLUDE-MODULES 215
  - in REPLACE-MODULES 271
  - in SAVE-LLM 37, 44, 288
  - in SHOW-LIBRARY-ELEMENT 307
  - in START-LLM-UPDATE 331
  - same 285
- element type L 10, 15, 16
  - in INCLUDE-MODULES 213
  - in REPLACE-MODULES 54, 268
  - in SAVE-LLM 43, 323
  - in START-LLM-UPDATE 329

- element type R 9
  - as BINDER input 140
  - in INCLUDE-MODULES 213
  - in REPLACE-MODULES 268
- element type, definition 23
- element version 48, 54
  - definition 23
  - highest 48, 54
  - in INCLUDE-MODULES 215
  - in REPLACE-MODULES 271
  - in SAVE-LLM 37, 44, 288
  - in SHOW-LIBRARY-ELEMENT 307
  - in START-LLM-UPDATE 331
- END 140, 168, 171, 212
- end of a sub-LLM 57
- END-LLM-STATEMENTS 57
- END-SUB-LLM-STATEMENTS 52, 209, 212, 262
- ENTRY 105
  - handling by autolink 277
  - renaming 107, 263
  - searching for 80
- entry point (ENTRY) 105, 107
  - renaming 263
  - searching for 80
- ENTRY-POINT 298
- error processing, controlling 134, 229
- ESV (External Symbols Vector) 29
  - in name conflicts 102
  - in SAVE-LLM 44, 294
  - when masking symbols 116
- exclusive slice 68
- EXIT-ROUTINE in SHOW-MAP 319
- external dummy section (XDSEC-D) 105
- external reference 105
  - renaming 107, 263
  - resolving 80, 277
  - resolving, examples 83
  - resolving, rules 81
  - unreferenced 80
  - unresolved 96, 299
  - unresolved, examples 96
- External Symbols Vector (ESV)
  - definition 29
  - in name conflicts 102
  - in SAVE-LLM 44, 294
  - when masking symbols 116
- EXTRN 80, 105, 299
  - renaming 107, 263
- F**
- FATAL ERROR 231
  - severity class 134
- filename (data type) 196
- fixed (data type) 195
- G**
- gen (suffix for data type) 206
- global index 204
- GLOBAL-INFORMATION 314
  - in BINDER lists 142
  - list 145
- guaranteed abbreviations 190
- H**
- handling symbols 105
- HELP-INFORMATION 314
  - in BINDER lists 142
- I**
- identification of an LLM 23
- INCLUDE-MODULES 30, 40, 48
  - format 213
- including modules 30, 48, 213
  - example 49
- INCLUSION-DEFAULTS
  - in MODIFY-LLM-ATTRIBUTES 237
  - in START-LLM-CREATION 38, 328
  - in START-LLM-UPDATE 332
- index 204
- INFORMATION 230
  - severity class 134
- input source for BINDER 48, 54, 213, 268
  - current 48
  - opening 140
- INPUT-INFORMATION 317
  - in BINDER lists 142
- inputs for BINDER 138

- integer (data type) 197
- interactive mode 15, 81
- INTERNAL ERROR (severity class) 134
- internal name
  - in MODIFY-LLM-ATTRIBUTES 43, 234
  - in SAVE-LLM 44, 288
  - in START-LLM-CREATION 37, 324
  - root in logical structure of an LLM 16, 23
- internal version
  - definition 23
  - in MODIFY-LLM-ATTRIBUTES 43, 235
  - in SAVE-LLM 44, 288
  - in START-LLM-CREATION 37, 325
- INTERNAL-NAME
  - in MODIFY-LLM-ATTRIBUTES 43, 234
  - in START-LLM-CREATION 324
  - in START-LLM-CrEATION 37
  - root in logical structure of an LLM 16, 23
- INTERNAL-VERSION
  - definition 23
  - in MODIFY-LLM-ATTRIBUTES 43, 235
  - in START-LLM-CREATION 37, 325
- ISAM keys 186
  - BINDER-MAPs 142
  - description 369
- J**
- job variables
  - for monitoring the BINDER run 169
- L**
- level
  - in logical structure of an LLM 16, 24, 57
- LIBRARY
  - in INCLUDE-MODULES 214
  - in REPLACE-MODULES 270
  - in RESOLVE-BY-AUTOLINK 278
  - in SAVE-LLM 287
  - in START-LLM-UPDATE 330
- library element 15
- link and load module (LLM) 9, 15, 16, 140, 141
  - definition 9
- link-edit 9
- linkage 9
- linkage editor BINDER 9
- linker BINDER 9
- list for symbolic debugging (LSD) 29, 43, 66
  - in INCLUDE-MODULES 218
  - in MODIFY-LLM-ATTRIBUTES 238
  - in MODIFY-MODULE-ATTRIBUTES 243
  - in REPLACE-MODULES 273
  - in RESOLVE-BY-AUTOLINK 281
  - in SAVE-LLM 44, 296
  - in START-LLM-CREATION 38, 328
  - in START-LLM-UPDATE 40, 333
- lists 298
  - COMMON list 153
  - duplicate list 317
  - duplicate symbol definitions list 156
  - duplicate symbols definitions list 142
  - global information 145
  - header information 142
  - header line 142, 314
  - help information 142, 144, 314
  - input information 142, 156, 317
  - ISAM keys 142
  - logical structure 142, 147, 314
  - merged modules 142
  - naming 142
  - output destination 309, 317
  - outputting 312
  - overview of the logical structure 314
  - physical structure 142, 149, 315
  - program map 142, 150, 315
  - pseudo-registers 317
  - pseudo-registers list 142, 155
  - sorted program map 316
  - sorted symbols definitions list 142, 154
  - statement list 142
  - unresolved definitions list 142
  - unresolved external references 316
  - unused modules 317
  - unused modules list 142, 155
- lists from BINDER 133, 141
  - output 141

- LLM (link and load module) 9, 15, 16, 140, 141
    - contents 29
    - creating 30, 37, 323
    - creating logical structure 57, 209, 212
    - creating logical structure, example 59
    - creating physical structure 68
    - creating physical structure, example 70
    - creating, example 39
    - current 43, 285
    - defining physical structure 69, 301
    - definition 9
    - identification 23
    - in BINDER lists 142
    - load address 104, 296
    - logical structure 16, 29
    - modifying attributes 30, 43, 233
    - modifying the structure 242
    - physical structure 18, 37, 235, 325
    - physical structure, diagrammatic representation 68
    - physical structure, example 23
    - reference address 104
    - saving 30, 37, 43, 285
    - saving, example 45
    - start address 298
    - updating 30, 40, 329
    - updating, example 41
  - load address 103
    - of an LLM 104, 296
  - load module 15
  - LOAD-ADDRESS 296
  - loader 9
  - loading 9
  - loading and starting BINDER 167
  - local relocation dictionary (LRLD)
    - definition 29
    - in SAVE-LLM 44
  - logging control 133
  - logical name of an LLM, changing 66
  - logical structure information for an LLM 38
    - in INCLUDE-MODULES 218
    - in MODIFY-LLM-ATTRIBUTES 43, 238
    - in REPLACE-MODULES 273
    - in RESOLVE-BY-AUTOLINK 280
  - in SAVE-LLM 295
  - in START-LLM-CREATION 328
  - in START-LLM-UPDATE 332
  - logical structure of an LLM 16, 29, 59
    - creating 57, 209, 212
    - level 16, 57
    - modifying 66
    - root 16, 57
    - structure tree 16, 57
  - LOGICAL-STRUCTURE
    - in BINDER lists 142
    - in INCLUDE-MODULES 218
    - in MODIFY-LLM-ATTRIBUTES 43, 238
    - in REPLACE-MODULES 273
    - in RESOLVE-BY-AUTOLINK 280
    - in SAVE-LLM 44, 295
    - in SHOW-MAP 314
    - in START-LLM-CREATION 38, 328
    - in START-LLM-UPDATE 40, 332
  - LRLD (local relocation dictionary)
    - definition 29
    - in SAVE-LLM 44
  - LSD (list for symbolic debugging) 29, 44
    - in INCLUDE-MODULES 218
    - in MODIFY-LLM-ATTRIBUTES 43, 238
    - in MODIFY-MODULE-ATTRIBUTES 66, 243
    - in REPLACE-MODULES 273
    - in RESOLVE-BY-AUTOLINK 281
    - in SAVE-LLM 296
    - in START-LLM-CREATION 38, 328
    - in START-LLM-UPDATE 40, 333
    - storing 29
- M**
- main memory resident (RESIDENT) 19, 75, 114, 253
  - man (suffix for data type) 206, 207
  - mandatory (suffix for data type) 207
  - MAP 133, 141, 298
  - MAP-NAME 142
  - masked symbol 107, 366
  - masking of symbols 102
    - modifying 116, 259
  - MAX-ERROR-WEIGHT 231

- MERGE-MODULES 119, 223
  - examples 122
- MERGED-MODULES in BINDER lists 142
- merging modules 119, 223
- message handling 135
- MESSAGE-CONTROL 135, 230
- metasyntax of SDF 191
- MODE 301
- MODIFY-ERROR-PROCESSING 134, 135, 229
- MODIFY-LLM-ATTRIBUTES 23, 30, 43, 233
- MODIFY-MAP-DEFAULTS 133, 239, 298, 312
- MODIFY-MODULE-ATTRIBUTES 66, 242
- MODIFY-SYMBOL-ATTRIBUTES 114, 253
- MODIFY-SYMBOL-TYPE 118, 257
- MODIFY-SYMBOL-VISIBILITY 107, 116, 259
- modifying
  - attributes of an LLM 30, 43, 233
  - attributes of modules 66, 242
  - attributes of symbols 114, 253
  - attributes of symbols, example 114
  - default values for output of lists 239
  - logical structure of an LLM 66
  - masking of symbols 116, 259
  - physical structure 43
  - structure of an LLM 242
  - symbol types 118
  - type of external references 257
- module 16, 140
  - including 30, 48, 213
  - including, example 49
  - modifying attributes 66
  - modifying attributes 242
  - priority 216, 272, 278, 308
  - removing 52, 262
  - removing, example 53
  - replacing 30, 54, 268
  - replacing, example 55
- MODULE-CONTAINER
  - in INCLUDE-MODULES 214, 270, 287
  - in START-LLM-UPDATE 330
- multiple access to LLMs 140, 288, 329
- N**
- NAME 262
  - in INCLUDE-MODULES 217
  - in REPLACE-MODULES 269
- name (data type) 197
- name conflicts 102
  - in MODIFY-MODULE-ATTRIBUTES 66
  - in MODIFY-SYMBOL-VISIBILITY 116
  - in RENAME-SYMBOLS 107
- NAME-COLLISION in MODIFY-MODULE-ATTRIBUTES 66
- nesting with sub-LLMs 57
- NEW-NAME 266
  - in REPLACE-MODULES 272
- node 16, 24, 57, 209
- nonmasked symbol 366
- notational conventions for SDF 191
- O**
- object module (OM) 9, 16, 140
  - prelinked 10
- object module library (OML) 9
  - in INCLUDE-MODULES 48, 213
  - in REPLACE-MODULES 54, 268
  - input source for BINDER 140
- odd (suffix for data type) 206
- OM (object module) 9
- OML (object module library) 9
  - in INCLUDE-MODULES 48, 213
  - in REPLACE-MODULES 54, 268
  - input source for BINDER 140
- opening, input source 140
- OUTPUT 309, 317
- output destinations for BINDER lists 133, 141
- outputting lists from BINDER 141, 312
- overlays with slices 20
- OVERWRITE in SAVE-LLM 43
- overwriting, LLM as library element 43

**P**

parameter area, BINDER macro 173  
 path name 48, 57, 209, 262  
   abbreviation 25  
   definition 24  
   in INCLUDE-MODULES 217  
   in MERGE-MODULES 224  
   in MODIFY-MODULE-ATTRIBUTES 243  
   in RESOLVE-BY-AUTOLINK 86, 280  
   of a module, changing 66  
   structure 24  
 PATH-NAME  
   in BEGIN-SUB-LLM-STATEMENTS 209  
   in INCLUDE-MODULES 217  
   in MERGE-MODULES 223  
   in MODIFY-MODULE-ATTRIBUTES 242  
   in REMOVE-MODULES 262  
   in REPLACE-MODULES 270  
   in RESOLVE-BY-AUTOLINK 86, 280  
 physical structure information of an LLM 29  
 physical structure of an LLM 18, 37, 235, 325  
   creating 68  
   creating, example 70  
   defining 69, 301  
   diagrammatic representation 68  
   diagrammatic representation, example 69  
   example 23  
 physical structure, modifying 43  
 PHYSICAL-STRUCTURE 315  
   in BINDER lists 142  
 positional operands 191  
 posix-filename (data type) 197  
 posix-pathname (data type) 198  
 prelinked module 15, 16  
 prelinked object module 10  
 priority of modules 48, 216, 272, 278, 308  
 product-version (data type) 198  
 program 15  
 program definition 105, 263  
 program information for BINDER run 169  
 program library 10, 15  
   current 43  
   in INCLUDE-MODULES 48, 213  
   in REPLACE-MODULES 54, 268

  in SAVE-LLM 37, 43  
   in START-LLM-UPDATE 329  
   input source for BINDER 140  
 PROGRAM-MAP 315  
   in BINDER lists 142  
 PSEUDO-REGISTER 317  
   in BINDER lists 142  
 pseudo-register 104  
 pseudo-register vector 104  
 PUBLIC 19, 75, 114, 253

**Q**

quotes (suffix for data type) 207

**R**

read access (READ-ONLY) 19, 75, 114, 253  
 READ-ONLY 19, 75, 114, 253  
 Readme file 13  
 RECOVERABLE ERROR 231  
   severity class 134  
 reference 105, 263  
 reference address 104  
   of an LLM 104  
 region 69  
 relocation dictionary (RLD)  
   in SAVE-LLM 295  
 relocation information 29, 44  
 relocation of addresses 104  
 RELOCATION-DATA in SAVE-LLM 44  
 REMOVE-MODULES 40, 52, 262  
 removing modules 52, 262  
   example 53  
 RENAME-SYMBOLS 107, 263  
 renaming symbols 107, 263  
   examples 108  
 REPLACE-MODULES 30, 40, 54  
   format 268  
 replacing modules 30, 54, 268  
   example 55  
 residence mode 19, 75, 114, 253  
 RESIDENT 19, 75, 114, 253  
 RESOLUTION 300

- RESOLUTION-SCOPE
  - in BEGIN-SUB-LLM-STATEMENTS 210
  - in INCLUDE-MODULES 219, 274
  - in MODIFY-MODULE-ATTRIBUTES 244
  - in RESOLVE-BY-AUTOLINK 281
- RESOLVE-BY-AUTOLINK 80, 85, 277
- resolving external references 80, 277
  - BINDER, examples 82, 83
  - BINDER, rules 81
- return code indicator
  - for BINDER run 169
- return code, BINDER macro 174
- RLD (relocation dictionary)
  - in SAVE-LLM 295
- RMODE 19, 75, 114, 253
- root 16, 24
- root slice 68, 69, 218, 273
- RUN-TIME-VISIBILITY
  - in MODIFY-MODULE-ATTRIBUTES 66
  - in RESOLVE-BY-AUTOLINK 86
- runtime modules
  - in RESOLVE-BY-AUTOLINK 86
- S**
- SAVE-LLM 30, 37
  - after START-LLM-CREATION 323
  - after START-LLM-UPDATE 40, 43, 329
  - for BINDER lists 133
  - format 285
- saving an LLM 30, 43, 285
  - example 45
- SCOPE
  - in MODIFY-SYMBOL-ATTRIBUTES 114, 254
  - in MODIFY-SYMBOL-TYPE 258
  - in MODIFY-SYMBOL-VISIBILITY 116, 260
  - in RENAME-SYMBOLS 266
  - in RESOLVE-BY-AUTOLINK 279
  - in SET-EXTERN-RESOLUTION 300
- scope for
  - BINDER run 138
  - edit run 138
  - one statement 138
- search procedure for path name 25
- security component BLSSEC 10
- sep (suffix for data type) 206
- SET-EXTERN-RESOLUTION 96, 299
- SET-USER-SLICE-POSITION 20, 69, 236, 301, 326
- severity class 170, 229
  - FATAL ERROR 134, 231
  - for messages 135
  - INFORMATION 134, 230
  - INTERNAL ERROR 134, 231
  - RECOVERABLE ERROR 134, 231
  - SYNTAX ERROR 134, 231
  - UNRESOLVED EXTERNS 134, 231
  - WARNING 134, 230
- shareable (PUBLIC) 19, 75, 114, 253
- SHOW-FILE 133, 141, 309, 318
- SHOW-LIBRARY-ELEMENTS 305
  - ISAM keys of the lists 131, 142
- SHOW-MAP 69, 133
  - format 312
  - modifying default values 239
  - output destination 141
- SHOW-SYMBOL-INFORMATION 321
- simultaneous access to LLMs 140, 288, 329
- simultaneous BINDER runs 140
- single slice 18, 21, 235, 325
  - common promotion 103
- SLICE 218, 273
  - slice 18, 218, 273, 301
    - by attributes 235, 325
    - by attributes, example 21
    - exclusive 68
    - level number 69
    - region 69
    - single 18, 235, 325
    - single, example 21
    - user-defined 20, 236, 326
    - user-defined, example 21
  - slice name 77, 149
- SLICE-DEFINITION 37, 69, 235, 325
  - in MODIFY-LLM-ATTRIBUTES 43
- SLICE-NAME 301
- slices by attributes 18



SORTED-PROGRAM-MAP 316  
     in BINDER lists 142  
 SPECIAL-HANDLING 231  
 standard header 174  
 start address of LLM 298  
 START-LLM-CREATION 23, 30, 37, 40  
     format 323  
 START-LLM-UPDATE 30, 40  
     format 329  
 START-STATEMENT-RECORDING 334  
 starter 10  
 STATEMENT-LIST  
     in BINDER lists 142  
 statements for BINDER 138, 187  
     grouping by function 187  
     overview 208  
 status indicator for BINDER run 169  
 STOP-STATEMENT-RECORDING 335  
 structure information for an LLM 29  
     logical, with SAVE-LLM 44  
     structure information, physical 29  
     structure of an LLM modifying 242  
     structure of path name 24  
     structured-name (data type) 198  
 sub-LLM 15, 16, 48, 52, 209  
     beginning 57, 209  
     end 57, 212  
     nesting 57  
 subsystem 19  
     symbols 236, 326  
 SUBSYSTEM-ENTRIES 236  
 suffixes for data types 192, 201  
 symbol  
     definition of term 105  
     handling 105  
     masked 107, 259, 366  
     modifying attributes 114, 253  
     modifying masking 116, 259  
     modifying type 118  
     nonmasked 366  
     renaming 107, 263  
     renaming, examples 108  
     showing information about 321  
 symbol types 257

SYMBOL-DICTIONARY  
     in SAVE-LLM 294  
 SYMBOL-DICTIONARY in SAVE-LLM 44  
 SYMBOL-NAME 253, 259, 264, 279, 299  
     in SHOW-LIBRARY-ELEMENT 308  
 SYMBOL-TYPE 264, 299  
 symbols in subsystems 236, 326  
 syntax description 191  
 SYNTAX ERROR, severity class 134, 231  
 SYSPLAMALT 165  
     FILE ID 165, 166

## T

task switch 231  
 temp-file (suffix for data type) 206  
 terminating BINDER 167  
 termination code for BINDER run 169  
 TEST-OPTIONS in REPLACE-MODULES 273  
 TEST-SUPPORT  
     in INCLUDE-MODULES 218  
     in MODIFY-LLM-ATTRIBUTES 43, 238  
     in MODIFY-MODULE-ATTRIBUTES 66, 243  
     in RESOLVE-BY-AUTOLINK 281  
     in SAVE-LLM 44, 296  
     in START-LLM-CREATION 38, 328  
     in START-LLM-UPDATE 40, 333  
 text (data type) 198  
 text information (TXT) 29  
 time (data type) 198  
 TXT (text) 29  
 TYPE 216, 272, 278, 308  
     in INCLUDE-MODULES 48

## U

unreferenced EXTRNs 118  
 unresolved external references 96, 299  
     examples 96  
 UNRESOLVED EXTERNS  
     severity class 134  
 UNRESOLVED-EXTERNS 231  
 UNRESOLVED-LIST 316  
     in BINDER lists 142  
 UNUSED-MODULE-LIST 317  
     in BINDER lists 142

updating LLM 30, 40, 329  
  example 41  
user (suffix for data type) 207  
user interface, notes on 190  
user switch 231  
USER-COMMENT 142, 314

### V

V-type constant 105, 107, 263, 299  
vers (suffix for data type) 207  
VERSION  
  in INCLUDE-MODULES 215  
  in REPLACE-MODULES 271  
  in SAVE-LLM 23  
  in START-LLM-UPDATE 331  
VISIBLE 260  
vsn (data type) 199

### W

WARNING 231  
WARNING-MESSAGE 265  
WARNING, severity class 134, 230  
weak external reference (WXTRN) 80, 105, 107,  
  263, 299  
wild-constr (suffix for data type) 204  
wild(n) (suffix for data type) 202  
with-constr (suffix for data type) 204  
with-wild(n) (suffix for data type) 202  
without (suffix for data type) 206  
without-cat (suffix for data type) 206  
without-corr (suffix for data type) 206  
without-gen (suffix for data type) 206  
without-man (suffix for data type) 206  
without-odd (suffix for data type) 206  
without-sep (suffix for data type) 206  
without-user (suffix for data type) 207  
without-vers (suffix for data type) 207  
work area of BINDER 37, 40, 43, 48, 329  
  deleting 323  
WXTRN 105  
  in RENAME-SYMBOLS 107  
  in SET-EXTERN-RESOLUTION 299  
  renaming 263  
  unresolved 80

### X

x-string (data type) 199  
x-text (data type) 199  
XDSEC-D 105