English

Fujitsu Software B2000

BLSSERV

Dynamic Binder Loader/Starter in BS2000

User Guide

Valid for:
BLSSERV V21.0A

Edition June 2021

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: bs2000.info@fujitsu.com.

Certified documentation according to DIN EN ISO 9001:2015

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 9001:2015.

Copyright and Trademarks

Table of Contents

BLSSERV **6**
1 Preface **7**
1.1 Brief description of the Binder Loader System **8**
1.2 Target group **11**
1.3 Summary of contents **12**
1.4 Changes from version 2.8 compared to the previous version 2.7 . . . . . **13**
1.5 Changes from version 2.7 compared to the version 2.5 . . . . . **14**
1.6 Metasyntax **15**
1.6.1 Notational conventions 16
2 The dynamic binder loader DBL **17**
2.1 Linking and loading **18**
2.1.1 Input to DBL 19
2.1.2 Link and load modules 20
2.1.3 Object modules 21
2.1.4 Primary and secondary input 22
2.1.5 Linkage process 23
2.1.6 Searching the primary input 24
2.1.7 Resolving external references 27
2.1.8 Handling COMMON sections 31
2.1.9 Address relocation 32
2.1.10 Support of EEN names (extended external names) 34
2.1.11 Handling name conflicts 35
2.1.12 Indirect linkage 37
2.1.13 Management of list name units 39
2.1.14 Loading LLMs from files (PAM-LLMs) 42
2.1.15 Output from DBL 43
2.1.16 Load unit 44
2.1.17 DBL map 46
2.1.18 Shareable programs (shared code) 50
2.1.19 System shared code 52
2.1.20 User shared code 53
2.1.21 LLMs with PUBLIC slices 54
2.2 Context concept **55**
2.2.1 Context as a set of objects 56
2.2.2 Context as a linking and loading environment 57
2.2.3 Context as an unloading and unlinking environment 58
2.2.4 Properties of a context 59

2.2.5 Example of the use of contexts 60
2.3 Additional functions 62
2.3.1 Unloading and unlinking objects 63
2.3.2 Outputting link and load information 64
2.3.3 Creating a separate symbol table 65
2.3.4 Processing REP files 66
2.4 Execution of the dynamic binder loader 68
2.4.1 Calling DBL 69
2.4.2 Message handling 71
2.5 Commands 72
2.6 Macros 73
3 XS support for DBL 74
3.1 Determining the addresses passed by the user 75
3.2 Pseudo-RMODE of an object module 76
3.3 Pseudo-RMODE of an LLM or PAM-LLM 77
3.4 Determining the load address 78
3.4.1 LOAD- and START-EXECUTABLE-PROGRAM (or LOAD- and START-PROGRAM) commands 79
3.4.2 BIND macro 81
3.4.3 Handling COMMON sections 83
3.5 Resolving external references above 16 Mb 84
4 The loader ELDE 85
4.1 ELDE functions 86
4.2 Calling ELDE 87
5 Migration 88
5.1 Existing concepts and new concepts 89
5.1.1 Existing concepts 90
5.1.2 New concepts 91
5.2 Features of the new Binder-Loader-Starter system 92
5.2.1 Link and load module (LLM) and BINDER 93
5.2.2 Dynamic binder loader DBL 94
5.2.3 DBL and BINDER 95
5.3 Migration from the dynamic linking loader DLL to DBL 96
5.3.1 Operating modes RUN-MODE=*STD and RUN-MODE=*ADVANCED 97
5.3.2 Incompatibilities in RUN-MODE=*ADVANCED 98
5.3.3 Loading LLMs in RUN-MODE=*STD 99
5.3.4 Loading an LLM with user-defined slices 100
5.3.5 Migration from the old to the new macros 101
5.4 Migrating from the static loader ELDE to the DBL 102
5.5 Migration of programs to PAM-LLMs 103
6 High-performance loading of programs and products 104

6.1 General notes on improving loading speed 105
6.2 Structural measures for reducing resource requirements 106
6.3 Improving the load speed for C programs 108
6.4 Use of DAB 109
7 Glossary 110
8 Related publications 117

BLSSERV

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 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 compiler at compilation time. This procedure is known as link editing (or binding).

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

1.1 Brief description of the Binder Loader System

The **Binder Loader System (BLS)** provides the user with the following functional units:

- the linkage editor BINDER
- the old linkage editor TSOSLNK
- the subsystem BLSSERV with the functionality of the dynamic binder loader DBL and the static loader ELDE
  BLSSERV V21.0A runs on BS2000/OSD-BC V6.0 or higher.
- the security component BLSSEC (which may be optionally activated).

The linkage editor 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) generated by compilers and stored in an object module library (OML), a program library (type R) or the temporary EAM object module file
- prelinked LLMs, or LLMs generated by compilers, from a program library (type L)
- prelinked LLMs from a PAM file (PAM-LLM)
- prelinked object modules linked by the TSOSLNK linkage editor and stored in an object module library (OML), in a program library (type R) or in the temporary EAM object module file.

The linkage editor TSOSLNK

The TSOSLNK linkage editor links:

- one or more object modules (OMs) into an executable program (load module) and stores this in a cataloged program file or as a type C library element in a program library
- multiple object modules (OMs) into a single prelinked module and stores this as a type R library element in a program library or in the EAM object module file.

Instead of the linkage editor TSOSLNK, the user should use BINDER, since TSOSLNK will not be developed further and will be replaced by BINDER.

BLSSERV with the dynamic binder loader DBL and the static loader ELDE

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,
- prelinked object modules linked by the TSOSLNK linkage editor and stored in an object module library (OML), in a program library (type R) or in the temporary EAM object module file.

The **static loader ELDE** loads an executable program that has been linked by TSOSLNK and stored in a program file or as a type C library element in a program library. The ELDE functionality is part of the BLSSERV subsystem.

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 or ELDE 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 by the individual functional units.The figure 1 shows the interaction between these functional units.

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

The linkage editors BINDER and TSOSLNK are utility routines. The dynamic binder loader DBL and the static loader ELDE, in contrast, belong to the subsystem BLSSERV which is a component of the BS2000 Control System. They offer their 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.

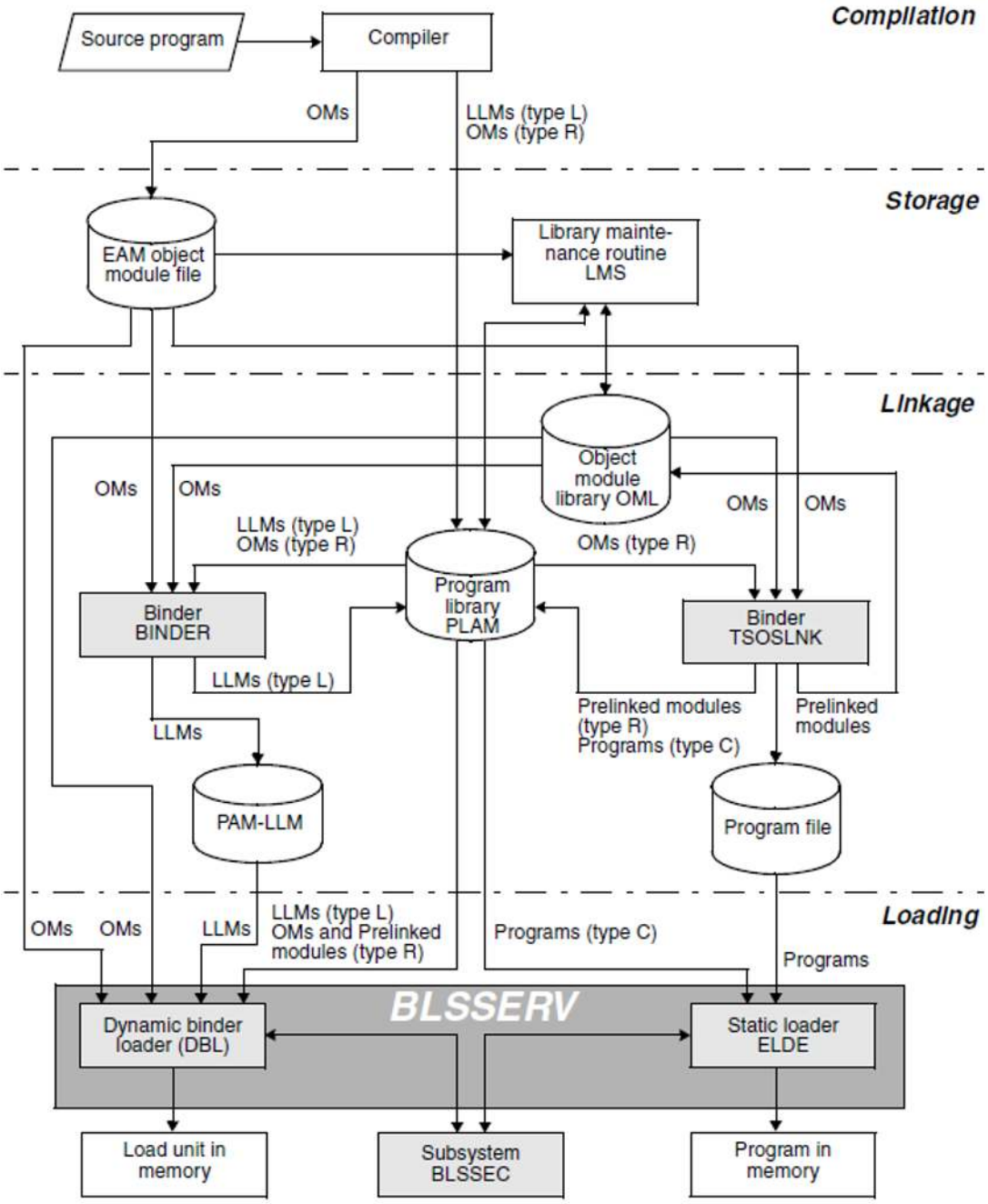


Figure 1: Interaction of the functional units for linking and loading

1.2 Target group

The “Dynamic Binder Loader / Starter” manual is addressed to the software developer. The manual provides a description of the facilities and use of the dynamic binder loader DBL and of the static loader ELDE as components of BLSSERV, and is also intended to serve as a reference work for their commands and macros.

1.3 Summary of contents

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

The present manual describes the dynamic binder loader DBL and the static loader ELDE.

The User Guide “BINDER” [1] contains the description of the linkage editor BINDER.
A separate Ready Reference is also available for BINDER, containing the formats of the BINDER statements.

This manual is organized as follows:

- The first chapter provides an overview of the functional units of the Binder-Loader-Starter (BLS) system and their interaction.
- The second chapter contains a description of the dynamic binder loader with its inputs and outputs, the restrictions for shareable programs, an explanation of the context concept, additional functions of DBL, execution of DBL, and an overview of the commands and macros used for calling DBL.
- The third chapter describes the XS support for the dynamic binder loader.
- The fourth chapter contains a short description of the static loader ELDE.
- The chapter “Migration” lists the main differences between the old linkage editor/loader concept (used up to BS2000 V9.5) and the actual Binder-Loader-Starter system introduced with BS2000 V10.0 and is intended to help the user make the transition.
- The chapter “High-performance loading of programs and products” contains notes on improving loading speed and describes measures for reducing resource requirements.

README file

Information on functional changes and additions to the current product version described in this manual can be found in the product-specific README file. You will find the README file on your BS2000 computer under the file name SYSRME.BLSSERV.*version*.*language*.

The user ID under which the README file is cataloged can be obtained from your systems support staff. You can view the README file using the /SHOW-FILE command or an editor, and print it out on a standard printer using the following command:

`/PRINT-DOCUMENT` *filename*`, LINE-SPACING=*BY-EBCDIC-CONTROL`

1.4 Changes from version 2.8 compared to the previous version 2.7

The current version of the “Binder Loader/Starter” manual incorporates the following changes compared to the previous version (“BLSSERV V2.7, Binder Loader/Starter”):

The following commands have been changed to BLSSERV 2.8A:

- LOAD-EXECUTABLE-PROGRAM
- START-EXECUTABLE-PROGRAM

Changed commands

LOAD-EXECUTABLE-PROGRAM

- The possible values for the sub-operands MINIUM and MAXIMUM of the RESIDENT-PAGES operand have been increased. Command description:

  /LOAD-EXECUTABLE-PROGRAM

  ...

  ,RESIDENT-PAGES = [*PARAMETERS](...)

  [*PARAMETERS](...)

  MINIMUM = *STD / <integer 0..2147483647 4Kbyte>
  ,MAXIMUM = *STD / <integer 0..2147483647 4Kbyte>

  ...

  Except for the possible values, the description of the operands has not changed.

START-EXECUTABLE-PROGRAM

- The possible values for the sub-operands MINIUM and MAXIMUM of the RESIDENT-PAGES operand have been increased. Command description:

  /START-EXECUTABLE-PROGRAM

  ...

  ,RESIDENT-PAGES = [*PARAMETERS](...)

  [*PARAMETERS](...) |

  MINIMUM = *STD / <integer 0..2147483647 4Kbyte>
  ,MAXIMUM = *STD / <integer 0..2147483647 4Kbyte>

  ...

  Except for the possible values, the description of the operands has not changed.

1.5 Changes from version 2.7 compared to the version 2.5

The current version of the “Binder Loader/Starter” manual incorporates the following changes compared to the previous version (“BLSSERV V2.5, Binder Loader/Starter”):

- BLSSERV V2.7 supports X86 code.
- Additional information is contained in the DBL list:
  - for LLMs and object modules (OMs): library name, element name and element version
  - for PAM LLMs: complete file name
- With BIND macro the external references from earlier load operations which remained unresolved can be output to a user-defined data area as well as those from the current load operation.
- With the ASHARE and ILEMGT macros the maximum number of parallel memory pools for shared code has been increased to 16.
- The redundant command and macro descriptions have been removed from this manual. The commands are described in detail in the "Commands" manual [5], and the macros in the "Executive Macros" manual [7].

1.6 Metasyntax

- Notational conventions

1.6.1 Notational conventions

The following notational conventions are used in this manual:

> ¡ for notes

boldprint for important concepts in descriptive text

Runtime examples are shown in `typewriter script.` Inputs expected from the user are highlighted in **`bold`**.

In the text, references to other publications are given as abbreviated titles with numbers enclosed in square brackets [ ]. The complete title of each publication to which reference is made is listed in “Related publications” at the back of the manual.

2 The dynamic binder loader DBL

The dynamic binder loader (DBL) is part of the BLSSERV subsystem. BLSSERV is loaded immediately after startup. The functionality of DBL is thus available via the program interface and a number of BS2000 commands. For this reason, DBL cannot be called like a utility routine.
The main function of DBL is to link modules into a load unit and load this into main memory. For this purpose, it is necessary to create for the module to be loaded an environment which permits all external references in this module to be resolved. This is done with the aid of the context concept of DBL. DBL also supports the shared use of modules by providing functions for loading and unloading shareable programs (shared code).

In addition to linking and loading, DBL performs the following functions:

- unloading and unlinking objects
- output of link and load information.

2.1 Linking and loading

Linking and loading is performed in the following steps (see figure 2):

- input to DBL
- linkage
- output from DBL

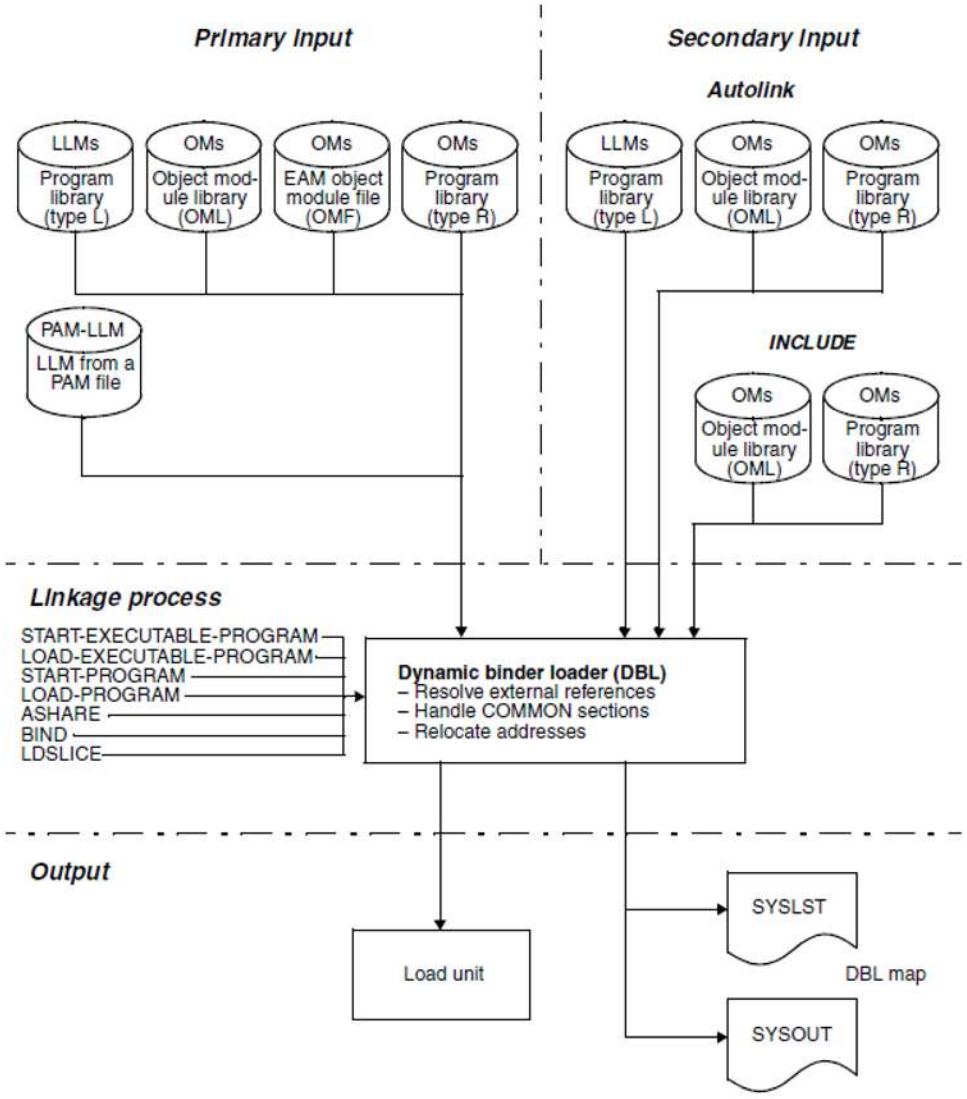


Figure 2: Linking and loading

2.1.1 Input to DBL

Input to DBL may consist of the following **modules**:

- link and load modules (LLMs) linked by BINDER or generated by compilers
- object modules (OMs) and prelinked object modules.

2.1.2 Link and load modules

Link and load modules (LLMs) are generated by the linkage editor BINDER and are stored as type L elements in program libraries or in PAM files. LLMs in PAM files are designated as PAM-LLMs. Compilers (such as C compilers) can also generate LLMs and then store them in program libraries (see the manuals for the compilers for details).

The logical structure of an LLM is implemented in the form of a tree. The root of this tree is located in the internal name of the LLM with which BINDER identifies the LLM. Nodes are sub-LLMs which are arranged hierarchically in levels. Object modules form the lowest level (leaves) of the logical structure of an LLM. The element name and element version of the LLM stored as a library element (type L) are used for external identification of the LLM. In the case of PAM-LLMs, the name of the PAM file is used for external identification.

In addition to its logical structure, an LLM also has a physical structure. The object modules of an LLM consist of control sections, the CSECTs, which can be loaded separately into main memory. The CSECTs of one or more OMs are combined into a slice. A slice is thus a loadable unit for DBL.

There are three types of physical structures for an LLM:

1. An LLM with a single slice (the entire LLM is loaded into memory in one load operation).
2. An LLM with slices by attributes:
   CSECTs which have the same attributes (READ-ONLY, RESIDENT, PUBLIC, RMODE) are combined into one slice (not permitted with PAM-LLMs).
3. An LLM with user-defined slices:
   The user specifies the positions of the slices. This permits the user to generate overlay structures (with PAM-LLMs up to 16 slices).

A detailed description of LLMs, their identification and their logical and physical structure can be found in the manual “BINDER” [1].

2.1.3 Object modules

Object modules (OM) are generated by compilers. Prelinked object modules that are generated by the TSOSLNK linkage editor have the same format as OMs and are therefore also treated as OMs in the following.

Input sources for modules may be:

- program libraries (PLAM, element type R)
- object module libraries (OML)
- the EAM object module file (OMF).

2.1.4 Primary and secondary input

A distinction is made between primary and secondary input, depending on the time the input takes place.

The **primary input** is specified in the command or macro with which DBL is called. The user specifies the module and, optionally, the input source in the command or macro. The module may be an object module or an LLM.

The **secondary input** is performed in the following cases:

- if DBL needs to include additional modules from an input source because of unresolved external references and V-type constants (autolink function, see "Resolving external references")
- if object modules contain INCLUDE statements that request additional object modules from an input source. This applies only to object modules. LLMs do not contain INCLUDE statements.

2.1.5 Linkage process

DBL links the modules of a load unit in the following steps:

- Primary input
  DBL searches for the module specified in the load call.
- Resolution of external references
  DBL tries to find matching names of control sections (CSECTs) and entry points (ENTRYs) for all external references in this module. For this, DBL searches for the appropriate modules in the secondary input and links these into the load unit. If the modules added in this manner result in further unresolved external references, this step is repeated iteratively until all external references have been resolved or until a previously declared status for unresolved external references is achieved.
- COMMON section processing
  DBL checks all modules for COMMON section definitions and reserves memory space for these.
- Address relocation
  All addresses contained in the modules are converted so that they refer to the same reference address.

These steps in the linkage process are described in detail below. Special features regarding the loading of list name units and PAM-LLMs are described in "Management of list name units" and in "Loading LLMs from files (PAM-LLMs)".

2.1.6 Searching the primary input

DBL searches for the module specified in the START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM command or the BIND macro in various “repositories”. The specified module name may be:

- the name of a CSECT or an ENTRY, which must not be masked
- the external name of an object module or an LLM
- the name of a program in the shared code in class 3/4/5 memory (loaded as anonprivileged subsystem)
- the name of a program in the user shared code.

The search is performed in the following stages:

1. Searching in the link context specified by the user (see "Context as a set of objects"). If nonmasked CSECTs or ENTRYs are found that match the specified symbol name in the load call, DBL passes the start address to the user task and the loading operation is terminated.
   - Searching in the link context is performed only for the BIND macro.
2. Searching in the user shared code which was loaded into the common memory pool with the DBL macro ASHARE. A start address is passed and the loading operation is terminated if DBL finds a shareable program (defined with the parameter PROGRAM in the ASHARE macro) or a nonmasked CSECT or an ENTRY with this name.
   - Searching in the user shared code is executed only in RUN-MODE=*ADVANCED.
   - Searching in the user shared code is **not** executed if NAME-SCOPE=*ELEMENT is specified in the START /LOAD-EXECUTABLE-PROGRAM command.
   - The common memory pools are searched only if the user specified SHARE-SCOPE= *MEMORY-POOL(...) /*ALL, since the default is SHARE-SCOPE= *SYSTEM-MEMORY. The search can be restricted to common memory pools with a defined scope or can be suppressed entirely (SHARE-SCOPE=*NONE).
3. Searching in the system shared code, which consists of the nonprivileged subsystems in class 3/4/5 memory (see the manual “Subsystem Management” [10]). If nonmasked CSECTs or ENTRYs are found that match the specified symbol name in the load call, DBL passes the load address to the user task and the loading operation is terminated.DBL thus establishes a connection between the user task and the subsystem.In RUN-MODE=*ADVANCED, the user can suppress searching in a nonprivileged subsystem by means of an operand in the load call (SHARE[-SCOPE]=*NONE).
   - Searching in the user shared code is **not** executed if NAME-SCOPE=*ELEMENT is specified in the START /LOAD-EXECUTABLE-PROGRAM command.
4. Searching in the library specified by the user in the load call (LIBRARY or LIBNAM@/LIBNAM/LIBLINK operand). If, in RUN-MODE=*ADVANCED, no such library was specified but a library was assigned to the link name BLSLIB, then DBL uses this default library with the link name BLSLIB. If the specified library is faulty or does not exist, processing is aborted with an error message.
5. Searching in

   alternate libraries.

   There are two groups of alternate libraries:

   a. Alternate libraries that are declared using file link names.

      In this case, it is necessary to distinguish between:

      - alternate user libraries with the file link names BLSLIBnn and

- alternate system libraries with the file link names $BLSLBnn
  Alternate system libraries are required for components of the runtime system supplied by Fujitsu. They are assigned independently by the respective runtime component by means of the file link name $BLSLBnn (00<=nn<=99) and cannot be influenced by the user. The file link names of alternate system libraries are managed by the Software Development of the Fujitsu Technology Solutions GmbH and are exclusively reserved for their software products.

b. System and user Tasklibs

   These are libraries with the name TASKLIB or a library that is assigned using the SET-TASKLIB command.

Alternate libraries declared by means of the file link name are searched in the following order:

a. Alternate system libraries with the file link name $BLSLB00 .. 49
b. Alternate libraries that were assigned by the user via the file link name BLSLIBnn (00<=nn<=99).
c. Alternate system libraries with the file link name $BLSLB50 .. 99 .

In a group of libraries these are searched in ascending numerical order (by “nn”).

System and user tasklibs are searched in the following sequence:

a. the library assigned by the user with the SET-TASKLIB command.
b. the “$userid.TASKLIB” library
   or, if this is not present,
   the “$defluid.TASKLIB” library
   Here, “defluid” is the name of the system default ID (value of the class 2 system parameter DEFLUID, see the “Introductory Guide to Systems Support” [9]).

Which of the alternate libraries is searched depends on the way in which the load operation is performed:

- Loading performed with START/LOAD-EXECUTABLE-PROGRAM
  or with the BIND macro (with INTVERS=SRVxxx, where xxx >= 002)

  Users can define which of the specified libraries are to be searched and in what order in the ALTERNATE-LIBRARIES or ALTLIB operand.
- Loading performed with START/LOAD-PROGRAM and RUN-MODE=*ADVANCEDor with the BIND macro (INTVERS=BLSP2/SRV001)

  Only the alternate libraries declared with the file link name BLSLIBnn or $BLSLBnn are searched if ALTERNATE-LIBRARIES=*YES has been specified.
- Loading performed with START/LOAD-PROGRAM and RUN-MODE=*STD

  Only the system and user Tasklibs are searched.

Selecting and assigning a program version

It is possible to select a program version to be used by DBL when searching for primary input and when resolving external references. As far as DBL is concerned, a program is basically a load unit with a specific version. All program definitions (CSECT, ENTRY, COMMON, ...) contained in the load unit inherit that version.

There are two ways of selecting a program version:

1. Selecting a version before loading and starting:

   The SELECT-PROGRAM-VERSION command or the SELPRGV macro is used to select a version of the load unit to be used by DBL in subsequent load calls. The load unit must not be already loaded on selecting the version, but multiple versions of that load unit may exist on the system.
2. Selecting a version in the load call:

The LOAD-EXECUTABLE-PROGRAM and START-EXECUTABLE-PROGRAM (or LOAD-PROGRAM and START-PROGRAM) commands or the BIND and ASHARE macros can be used to specify a version (with the operand PROGRAM-VERSION or PGMVERS, respectively) in the load call. DBL then searches among the already loaded objects for a load unit with the given name and exactly the same version. DSSM subsystems are also included in the search. DBL treats the subsystem version as a program version and takes all “connectable entries“ (see "System shared code") of the subsystem into account.

Note that a version selection in the load call will always override any previously made version selection.

If the selected version (see points 1 and 2 above) cannot be found among the loadedobjects, DBL assigns a version to the load unit that is to be loaded.

The selected version for a program can be queried by means of the GETPRGV macro.

The program version can also be selected for unloading and unlinking and for the output of link and load information.

2.1.7 Resolving external references

DBL tries to resolve all unresolved external references in the modules of a load unit, For this, it first searches the modules which are already linked for CSECTs or ENTRYs with the same name. If no matching symbols can be found, it searches for further modules containing such CSECTs or ENTRYs and includes these modules in the load unit. This loading operation is referred to as the **autolink function** of DBL. The autolink function handles only CSECTs and ENTRYs that are not masked (see the manual “BINDER” [1]).

Search strategy

DBL searches for the modules that resolve the external references in various “repositories”. If a module containing matching CSECTs or ENTRYs is found in one of the repositories, it is included in the load unit and the search operation is terminated.
By default, searching involves the following steps, and the sequence in which steps 1 to 4 take place can be changed by the user (see the note entitled "User defined searching order"):

1. Searching in the link context (see "Context as a linking and loading environment").
2. Searching in the user shared code which was loaded with the ASHARE macro.
   - Searching in the user shared code is executed only in RUN-MODE=*ADVANCED.
   - The common memory pools are searched only if the user specified SHARE-SCOPE= *MEMORY-POOL(...) /*ALL, since the default is SHARE-SCOPE=*SYSTEM-MEMORY. The search can be restricted to common memory pools with a defined scope or can be suppressed entirely (SHARE-SCOPE=*NONE).
3. Searching in the system shared code, which consists of the nonprivileged subsystems in class 3/4/5 memory (see the manual “Subsystem Management” [10]). If an external reference is resolved by a symbol of the subsystem, DBL establishes a connection to this subsystem. The user can prevent searches in a nonprivileged subsystem by specifying the appropriate operand in the load call (SHARE[-SCOPE]=*NONE).
4. Searching in reference contexts (see "Context as a linking and loading environment") which the user has defined. If several reference contexts exist, these are searched in their existing order. If no reference contexts exist, this step is skipped. This step is also possible only for the BIND macro.
5. Searching in libraries specified by the user in the load call (LIBRARY or LIBNAM@/LIBNAM/LIBLINK operand). If no such library was specified by the user (in RUN-MODE=*ADVANCED), but a library was assigned the link name BLSLIB, then this default library with the link name BLSLIB will be used by DBL. If the specified library is corrupt or does not exist, processing is aborted with an error message.
6. Searching in

   alternate libraries.

   There are two groups of alternate libraries:

   a. Alternate libraries that are declared using file link names.

      In this case, it is necessary to distinguish between:

      - alternate user libraries with the file link names BLSLIBnn and

- alternate system libraries with the file link names $BLSLBnn
  Alternate system libraries are required for components of the runtime system supplied by Fujitsu. They are assigned independently by the respective runtime component by means of the file link name $BLSLBnn (00<=nn<=99) and cannot be influenced by the user. The file link names of alternate system libraries are managed by the Software Development of the Fujitsu Technology Solutions GmbH and are exclusively reserved for their software products.

b. System and user Tasklibs

   These are libraries with the name TASKLIB or a library that is assigned using the SET-TASKLIB command.

Alternate libraries declared by means of the file link name are searched in the following order:

a. Alternate system libraries with the file link name $BLSLB00 .. 49
b. Alternate libraries that were assigned by the user via the file link name BLSLIBnn (00<=nn<=99).
c. Alternate system libraries with the file link name $BLSLB50 .. 99 .

In a group of libraries these are searched in ascending numerical order (by “nn”).

System and user tasklibs are searched in the following sequence:

a. the library assigned by the user with the SET-TASKLIB command.
b. the “$userid.TASKLIB” library
   or, if this is not present,
   the “$defluid.TASKLIB” library
   Here, “defluid” is the name of the system default ID (value of the class 2 system parameter DEFLUID, see the “Introductory Guide to Systems Support” [9]).

Which of the alternate libraries is searched depends on the way in which the loadoperation is performed:

- Loading performed with START/LOAD-EXECUTABLE-PROGRAM
  or with the BIND macro (with INTVERS=SRVxxx, where xxx >= 002)

  Users can define which of the specified libraries are to be searched and in whatorder in the ALTERNATE-LIBRARIES or ALTLIB operand.
- Loading performed with START/LOAD-PROGRAM and RUN-MODE=*ADVANCEDor with the BIND macro (INTVERS=BLSP2/SRV001)

  Only the alternate libraries declared with the file link name BLSLIBnn or $BLSLBnn are searched if ALTERNATE-LIBRARIES=*YES has been specified.
- Loading performed with START/LOAD-PROGRAM and RUN-MODE=*STD

  Only the system and user Tasklibs are searched.

Steps 5 and 6 above can be suppressed in the load call with START/LOAD-EXECUTABLE-PROGRAM or the BIND macro by specifying the operand AUTOLINK=*NO. If this is done, DBL will search only the already loaded private and shared code. This eliminates thepossibility of runtime modules being inadvertently loaded. Unresolved external references, if any, are handled by DBL as described in "Resolving external references".

If name conflicts are detected, DBL handles them as described in "Handling name conflicts".

The autolink function does *not* refer to the temporary EAM object module file. If modules are to be loaded from the EAM object module file (*OMF operand in the load call), the user must make sure that the first module in the file contains the start address of the load unit. In the load call, the user must specify that all modules are to be fetched from the EAM object module file (operand FROM-FILE=*MODULE(LIBRARY=*OMF,ELEMENT=*ALL,...)).

No overlay loading is initiated as a result of weak external references (WXTRNs). These are resolved only:

- in a nonprivileged subsystem or in the shared code
- by modules fetched, for example, using the INCLUDE statement
- by the autolink function called by other EXTRNs or V-type constants.

A COMMON section is not linked by the autolink function. This means that if an external reference refers to a COMMON section, the module containing this COMMON section cannot be loaded automatically. The external references are not resolved.

User defined searching order

Users can change the search sequence of steps 1 to 4 to suit their requirements. The following operands are available for this:

- For the MODIFY-DBL-DEFAULTS command:

  RESOL-TYPE=*USER(ORDER=...) and
  PUBLIC-RESOL-TYPE=*USER(ORDER=...)
- For the BIND macro:

  RESORD (with RESTYP=USER) and PURESOR (with PURESTY=USER)

Handling unresolved external references

The user can control how external references not resolved by searching the alternate libraries (see step 6 in "Resolving external references") are to be handled by including the UNRESOLVED-EXTRNS operand in the load call.

The following options are available:

- unresolved external references are not allowed
  (operand UNRESOLVED-EXTRNS=*ABORT)
  This causes loading of the current load unit to be aborted.
- unresolved external references are assigned a user-defined address(operand UNRESOLVED-EXTRNS=*STD)
  The address is specified by the user in the ERROR-EXIT operand in the load call. The default value for the address is X'FFFFFFFF'.
- unresolved external references are resolved at a later time
  (operand UNRESOLVED-EXTRNS=*DELAY)
  DBL stores the unresolved external references in the link context (see "Context as a linking and loading environment"). If a new load unit is loaded in the context, DBL tries to resolve the stored external references at the end of the loading operation using CSECTs and ENTRYs of the new load unit. This process is repeated when further load units are loaded, for as long as the context remains in existence. This option does not apply to external dummy sections (XDSEC-Rs). When stored in the context, external references that are to be resolved at a later time are given a temporary address, specified by the user in the load call by means of the ERROR-EXIT operand. The default value is *NONE, which internally corresponds to the address X'FFFFFFFF'.
- the specification UNRES=DELAYWARN in the BIND macro basicaly has the same effect as UNRES=DELAY. The two specifications differ only in the return code. If external references exist which must be resolved later, a corresponding return code is issued with DELAYWARN, while the return code with DELAY is null. In addition, the specification UNRES=DELAYWARN is a prerequisite for outputting external references from earlier load operations which remained unresolved to the user-defined data area.

When RUN-MODE=*STD is set, processing is performed according to the operand UNRESOLVED-EXTRNS=*STD, and all unresolved external references are given the address X'FFFFFFFF'.

DBL logs all unresolved external references in the SYSOUT system file. In interactive mode, with UNRESOLVED-EXTERNS=*STD set, the user can then decide whether processing is to continue or terminate when unresolved external references are detected.In batch mode, processing is always continued.

Unresolved dummy sections (XDSEC-Rs) are listed separately from the other external references. XDSEC-Rs are only resolved with XDSEC-Ds in the same context.

The USNUNR@ operand of the BIND macro (only in conjunction with INTVER=SRVxxx, where xxx >= 005) enables the output of a list of unresolved external references to a userdefined data area. This operand and the UNRES operand are totally independent of each other. Details are provided in the desription of the BIND macros in the „Executive Macros“ manual [7]. When INTVERS=SRVxxx (xxx >= 006) and UNRES=DELAYWARN are specified in the BIND macro, the list of external references from earlier load operations which remained unresolved can be output to the user-defined data area either as an alternative or in addition.

2.1.8 Handling COMMON sections

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 load unit as data communication areas between different modules or as reserved space for CSECTs.

COMMONs with the same name that are defined in different modules of the load unit are assigned a single shared memory area by DBL. DBL selects this area sufficiently large so that the longest COMMON section of this name can be accommodated. It is not possible to enlarge this memory area by a BIND macro call during the program run. If there is no control section (CSECT) with the same name as the COMMON section, DBL delays processing until all modules have been loaded. If a CSECT has the same name as a COMMON section, DBL assigns this CSECT the address of the COMMON section, i.e. the COMMON section is initialized with this CSECT when the load unit is loaded. Also, the attributes and contents of the COMMON section are matched to those of the CSECT during initialization.

If a CSECT has the same name as a COMMON section, DBL takes the COMMON statement into account earlier, namely when processing the relevant CSECT.

- If the COMMON section appears in the input ahead of the identically named CSECT, DBL assigns the greater of the two lengths to the COMMON.
- If the CSECT appears in the input ahead of the identically named COMMON section, DBL makes the COMMON the same length as the CSECT, regardless of the length specified in the COMMON definition.

The autolink function has no effect for COMMON sections. A module with a COMMON section referred to by an EXTRN or VCON is thus not automatically loaded.

If name conflicts occur, they are handled as described in "Handling name conflicts".

2.1.9 Address relocation

The most important function of DBL is to form an executable unit from the object modules (OMs) and LLMs of the input. It therefore has to adjust the addresses in the individual control sections to match the overall program (relocation) and also search for the matching entry addresses for any still unresolved external references and V-type constants.

An OM and an LLM contain relocation records which define how the program addresses relate to one another. This information is used by DBL to assign the definitive addresses for the overall program.

Address relocation for object modules (OMs)

The start address for CSECTs depends on the two attributes READ-ONLY and PAGE, entered by the compiler:

READ-ONLY means that the CSECT is to be read-only during the program run, i.e. it may not be overwritten. DBL must therefore load the CSECT into a new page if the current page does not also have the READ-ONLY attribute.

PAGE declares that the CSECT is to be loaded at the start of a new page, i.e. at an address that is a multiple of 4096 bytes (X'1000').

Examples of how control sections are processed by DBL

1.

```
X   CSECT   PAGE,READ
.....................
Y   CSECT   READ
.....................
```

   Control section X is to be aligned on a page boundary and be read-only. The next control section, Y, does not need to be page-aligned and is likewise read-only. DBL therefore loads Y into the same page as X, at the next doubleword boundary.

2.

```
ABC   CSECT   READ
..................
XYZ   CSECT
..................
```

   DBL automatically aligns control section ABC on a page boundary if it is the first CSECT of the object module. It loads ABC into a read-only page. Control section XYZ is aligned on the page boundary of a new page by DBL because read and write access are allowed for the CSECT.

Address relocation for LLMs

The start address is dependent on the following properties of the LLM:

1. LLM with CSECT attributes PAGE and READ-ONLY

   If the PAGE and READ-ONLY attributes have been passed by BINDER, the following applies:

   - PAGE specifies that the CSECT is to be loaded at the start of a new page, i.e. at an address that is a multiple of 4096 bytes (X'1000').

- READ-ONLY means that the CSECT may only be read during the program run, i.e. it must not be overwritten. DBL must therefore load the CSECT into a new page if the current page does not also possess the READ-ONLY attribute.
  This applies to LLMs with the following physical structure (see the “BINDER” manual [1]):
  - single slices
  - user-defined slices
  - slices by attributes, where the READ-ONLY attribute has not been used for creating the slices.

2. LLM without relocation information (LRLD)

   If the LLM has been stored without relocation information (see the BINDER statement SAVE-LLM, operand RELOCATION-DATA=*NO [1]), DBL determines the start address specified with the LOAD-ADDRESS operand in the SAVE-LLM statement. If this is not possible, the loading operation is aborted.

3. LLM with relocation information (LRLD)

   If the LLM has been stored with relocation information (see the BINDER statement SAVE-LLM, operand RELOCATION-DATA=*YES [1], DBL determines the start address as follows:

   - First, it tries to assign the address specified with the LOAD-ADDRESS operand in the BINDER statement SAVE-LLM as the start address. If this is possible, relocation is limited to the unresolved external references and to the references to COMMON sections. (Address relocation within the LLM was already done during the BINDER run.)
   - If the address defined with the LOAD-ADDRESS operand cannot be assigned as the start address, DBL selects an arbitrary address and performs a complete relocation.

2.1.10 Support of EEN names (extended external names)

BLSSERV supports LLMs with extended external names (EEN).

Compilers of object oriented programming languages generate symbol names that do not comply with the conventions for external names (ENs). Names of this kind are called EEN-Names (Extended external names).

Symbols with EENs can only be contained in LLMs with format 4 (or higher).

- These LLMs can be created by BINDER as of V2.0
- They can be loaded with BLSSERV as of V2.0 in the user address space.
- With BLSSERV as of V2.1 they can also be loaded as shared code or as a subsystem.
- Input of EEN names via user interface for binding and loading (START/LOAD-EXECUTABLE-PROGRAM, START/LOAD-PROGRAM, BIND, ASHARE) is not possible. The modules must be specified by the name of the library element which contains them.
- Information about EENs can be output via the VSVI1 interface. In all other BLS output no EENs will be displayed. They are replaced by internal names.

2.1.11 Handling name conflicts

Name conflicts can arise when symbols with identical names occur in the external symbol dictionary of a load unit. Symbols are CSECTs, ENTRYs, COMMONs and XDSEC-Ds. Note that *identical* names are not necessarily always *conflicting* names.

DBL handles name conflicts differently, depending on whether the operating mode is RUN-MODE=*STD or RUN-MODE=*ADVANCED (see "Operating modes RUN-MODE=*STD and RUN-MODE=*ADVANCED").

Handling name conflicts in RUN-MODE=*STD

In the case of RUN-MODE=*STD, name conflicts are detected regardless of whether or not the symbols in a module are masked. The user has no means of controlling how name conflicts are handled.

The following table shows how name conflicts are handled by DBL. “Entry 1” stands for a symbol in a loaded module or for a COMMON that is already known in the current load unit. Any uninitialized COMMON at the end of the load unit is flagged in the symbol table and treated as a CSECT in subsequent load calls. A name conflict occurs if an “Entry 2” symbol of the same name is found in the module to be loaded.

	Entry 1			
Entry 2	**CSECT**	**ENTRY**	**COMMON**	**XDSEC-D**
CSECT	(1)	---	(2)	--
ENTRY		---	(3)	--
COMMON	(3)	(4)	(5)	--
XDSEC-D	--	--	--	(6)

Explanation

(1) A name conflict has been detected.Loading of the module containing the CSECT with the same name is aborted.The autolink function is terminated. If loading of the module was initiated by an INCLUDE statement in an object module (see appendix), the module is ignored and processing continues.

(2) The COMMON section is initialized with the CSECT.

(3) A name conflict has been detected.Loading is aborted.

(4) The name conflict is **not** detected.

(5) The second COMMON section is ignored.

(6) Recoverable error.The second XDSEC-D is skipped.

(-) Not a name conflict (only an identical name).

Handling name conflicts in RUN-MODE=*ADVANCED

In RUN-MODE=*ADVANCED, name conflicts are detected only if the symbols in a module are *not* masked. The user is notified with a message of any name conflict which could lead to errors. By including the NAME-COLLISION operand in the load call, the user can control how name conflicts are handled. The following options are available:

- operand NAME-COLLISION=*STD
  Name conflicts between nonmasked symbols are indicated by warning messages. The module containing the symbol with the same name is loaded. The new occurrence of the symbol is masked, i.e. it is not used to resolve external references.

- operand NAME-COLLISION=*ABORT
  Loading of the current load unit is aborted as soon as a name conflict between nonmasked symbols is detected.

The following table shows how DBL handles name conflicts when the user has specified NAME-COLLISION=*STD. “Entry 1” stands for a nonmasked symbol in a loaded module or for a COMMON that is already known in the current load unit. Any uninitialized COMMON at the end of the load unit is flagged in the symbol table and treated as a CSECT insubsequent load calls. A name conflict occurs if a nonmasked “Entry 2” symbol of the same name is found in the module to be loaded.

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

Explanation

(1) A name conflict has been detected.The symbol “Entry 1” is not masked and can be used to resolve external references. The symbol “Entry 2” is masked and cannot be used for resolving external references. A warning is issued.

(2) The COMMON section is initialized with the CSECT (as for RUN-MODE=*STD).

(3) The second COMMON section is ignored (as for RUN-MODE=*STD).

(4) Recoverable error.The second XDSEC-D is skipped. The name conflict is handled in the same way as for RUN-MODE=*STD.

(-) Not a name conflict (only an identical name).

2.1.12 Indirect linkage

The new mechanism, which is referred to as “indirect linkage” below, was introduced in DBL. The term “indirect linkage” means that an external reference is not resolved directly with a program definition in the usual manner, but by an intermediate indirect linkage routine. This IL routine includes a call to a server module containing the required program definition.

Indirect linkage has the following advantages:

- When a server module is swapped, only the link between the IL routine and the server needs to be removed. A new server module can then be used for the same program definition simply by resolving the external reference between the IL routine and the server module. This eliminates the additional overhead for measures to ensure consistency between the server and its possible callers.
- The resolution of external references occurs only once and does not change so long as the caller is loaded. This eliminates unlinking across multiple contexts or tasks as well as the need to resolve external references repeatedly whenever the server is called.

Symbols of type ILE

A new symbol type ILE (Indirect Linkage Entry) with the attributes below was added for indirect linkage:

- Name:
  corresponds to the name of the ILE server
- Address of the IL routine
- Address of the ILE server:
  a field containing the ILE server address
- Offset of the server address in the IL routine:
  locates the field with the ILE server address within the IL routine
- Status of the server (active or inactive):
  indicates whether the ILE server can be currently called.
  The status can be used both by the calling program and by any user-defined IL routine.
- Control (by the system or the user)

The declaration of ILEs and the maintenance of ILE lists are performed by means of the ILEMIT and ILEMGT macros. Information on ILEs can be obtained by calling the VSVI1 macro.

The figure 3 illustrates how external references are resolved when the ILE server is loaded.

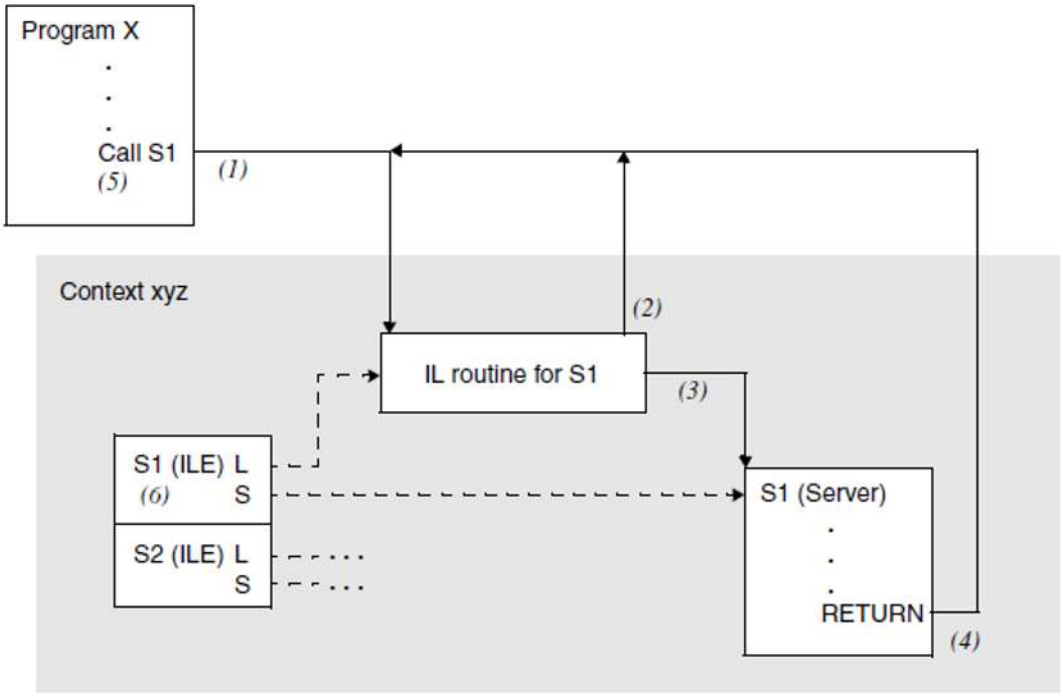


Figure 3: Indirect linkage

(1) The call to S1 branches to the IL routine for S1.

(2) If the server module S1 has not been loaded, a return to the calling program X is effected directly from the IL routine.

(3) If the server module S1 is loaded, the IL routine branches to the server module S1.

(4) After the server module S1 has been processed, control is returned to the calling program X.

(5) The caller can then check the return code to obtain information on the processing of the server module.

(6) The symbol table of the context xyz contains an entry for an ILE symbol named S1, in which the addresses of the IL routine (L) and the server module (S) are stored.

2.1.13 Management of list name units

As of BLSSERV V2.4A, several or all objects of a library can be loaded with a single call of the BIND macro. The result is a so-called list name unit, and multiple DBL calls whenloading a symbol list of the same load unit are prevented.

For loading into a list name unit the specifications `INTVERS=SRVxxx` (`xxx` >= 004) and `SYMBOL=*ALL` or `SYMBOL=name` are required, the last character of `name` being the wildcard symbol “*”. For details, see the BIND macro in the „Executive Macros“ manual [7].

Loading into a list name unit proceeds as follows:

- Depending on the SYMBOL operand, a list of the objects to be loaded from the main library is created:
  - With `SYMBOL=*ALL`, first the highest versions of all elements of type L are entered in this list. Then the highest versions of all elements of type R are entered in the list, but only if an element of type L with the same name is not already contained in the list.
  - If the SYMBOL operand is specified with a partially qualified name, the list is generated in the same way. However, only those elements are taken into account which match the partially qualified name.
  - The name list contains only names of PLAM library elements (i.e. SYMTYP=MODULE). No CSECTs or ENTRYs with the same name are searched for in the library.

  The following requirements apply here:
  - The main library must be a PLAM library.
  - Alternative libraries are only used for the autolink process, not for generating the name list.
  - LLMs with user-defined slices cannot be loaded into a list name unit. The load operation is aborted with RC=X'0C400430'.
  - The VERS and VERS@ operands are ignored when loading into a list name unit.
  - No names in the code already loaded are searched for before loading as these names are element names and not symbol names. However, only symbol names are stored in the loaded code.
- All objects in the list are loaded in the same unit (list name unit) and in the contexdefined by UNIT or UNIT@ and LNKCTX or LNKCTX@. If no name is specified for the load unit, the name of the first object in the list is assumed as the name for the load unit.
- The version of a list name unit is determined by the PGMVERS/PGMVERS@ operand. In the event of `PGMVERS=*STD` the PLAM version of the first object loaded is used as the version of the load unit.
- In the DBL map, EXPLICIT is specified as the linkage method for each object contained in the list.
- Both the returned start addess (SYMBLAD operand with BIND) and the returned AMODE correspond to those of the first object loaded into the list name unit. When `BRANCH=YES` is specified, control is also transferred to the start address of the first object loaded in the list name unit.

- When loading into a list name unit, external references are only resolved when all objects of the list name unit have been loaded. The autonlink function is also only executed at this point. This enhances performance when resolving external references: A symbol which is referenced in multiple modules of the list need only be searched for once and not after each individual object has been loaded.

  Example

  Three objects of a list are to be loaded:

  - Module M1 with external references E1 and E2
  - Module M2 with external references E2 and E3
  - Module M3 with ENTRY E1 and external reference E2

  1. Load operation when loading **without** list name unit:
     - Load M1
     - Resolve external references E1 and E2 and autolink (E1 found in M3)
     - Load M3 (because of autolink)
     - Resolve external reference E2 and autolink (no effect)
     - Load M2
     - Resolve external references E2 and E3 and autolink (no effect)

     The process of address resolution is executed three times for external reference E2.
  2. Load operation when loading **into** list name unit:
     - Load M1
     - Load M2
     - Load M3
     - External references E1, E2 and E3 and autolink (no effect)

     The process of address resolution is executed only once for external reference E2.

  Loading into a list name unit is thus more efficient, but can influence the load sequence (without a list name unit module M3 is loaded before M2).

  When loading from PUBLIC/PRIVATE LLMs into list name units, the public part is loaded only after all objects of the list name unit have been loaded.

The following restrictions apply for unloading list name units:

- A list name unit can only be unloaded in full. It is not possible to unload part or just a single module from a list name unit.
- If errors occur for one of the objects in the list, all modules of the list which have already been loaded (= current status of the list name unit) are unloaded. This corresponds to the current behavior when loading a unit.

Characteristics of a list name unit

The characteristics of a list name unit are summarized below:

- The input objects are only searched for in the main library (alternative libraries are only used for autolink).
- Input objects are always loaded (no search in the memory prior to loading).
- External references are resolved and the autolink function executed only when all objects of the list name unit have been loaded.
- A module of a list name unit cannot be unloaded independently of the list name unit.

2.1.14 Loading LLMs from files (PAM-LLMs)

To facilitate conversion of applications with load modules to those with LLMs for the user, BLSSERV enables LLMs to be loaded which were stored in a PAM file with BINDER. In contrast to the LLMs stored in program libraries, these LLMs are called PAM-LLMs.

PAM-LLMs differ from the conventional LLMs in the following ways:

- PAM-LLMs can only be loaded with commands, but not with program interfaces such as the BIND macro. (Exception: Slices of a PAM-LLM with user-defined slices can be loaded with the LDSLICE macro.)
- The file link name for PAM-LLMs is the same as for program files: ECERDLOD.
- As a PAM-LLM runs in ADVANCED execution mode, it is by default loaded in the LOCAL#DEFAULT context, not in the CTXPHASE context, which is reserved for load modules.
- The name in the load call is always interpreted as a file name and not as a symbol name. Consequently no search for symbols already loaded takes place in the memory during loading.
- During loading no external references are resolved and no autolink function isexecuted.
- PUBLIC/PRIVATE-PAM-LLMs cannot be loaded.

2.1.15 Output from DBL

Output produced by DBL may be:

- a load unit that is contained within a context and can be started
- a DBL map containing information about the logical structure and contents of the loaded load unit.

2.1.16 Load unit

When DBL is called with a START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM command or by means of a BIND or ASHARE macro, it generates a **load unit** which it loads into main memory. The structure of a load unit is shown in figure 4.

A load unit contains all the modules loaded with *one* load call. This means all the modules specified in the load call (primary input), plus any modules inserted by the autolink function or as a result of INCLUDE statements (secondary input).

The modules may be object modules (OMs) or LLMs. They contain the program definitions (CSECTs and ENTRYs). Each load unit is contained within a context that is defined by the user (see "Context concept"). If the user does not specify a context, the default context with the name “LOCAL#DEFAULT” is used.

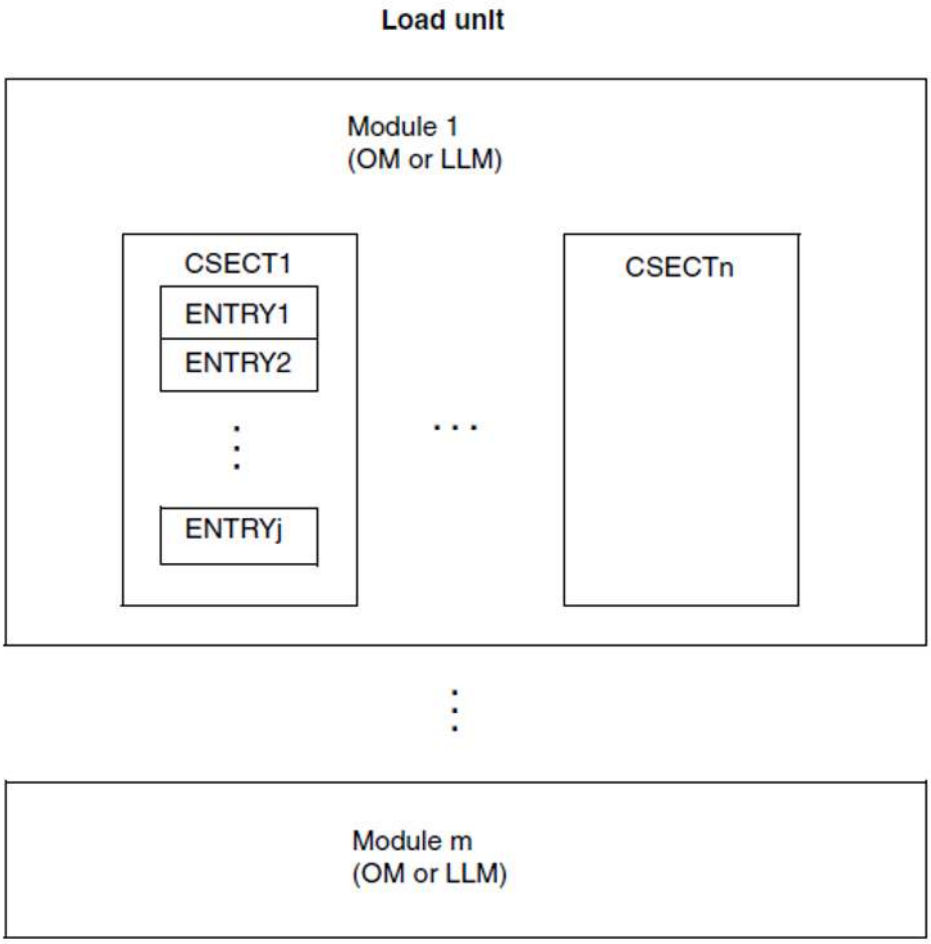


Figure 4: Structure of a load unit

Load information for a load unit

The user can define the load information for a load unit independently in the load call (using the LOAD-INFORMATION and TEST-OPTIONS operands). These enable the user to select whether or not the load unit is to contain an external symbol dictionary.
For LLMs, this information can be loaded only if it was created by BINDER (BINDER statement SAVE-LLM, operand SYMBOL-DICTIONARY=*YES).
For object modules (OMs), the load information can always be loaded.
For prelinked OMs, the load information may be reduced using the TSOSLNK linkage editor.
If the load unit is to include an external symbol dictionary, the following information can be selected for the load unit:

- External symbol dictionary containing program definitions
  (operand LOAD-INFORMATION=*DEFINITIONS)
  An external symbol dictionary containing the program definitions of all modules in the load unit is loaded.
  Program definitions are control sections (CSECTs), entry points (ENTRYs), COMMON sections, dummy sections (DSECTs), external dummy sections (XDSEC-Ds) and module names. DBL needs this information in order to
  - resolve external references between modules loaded as a result of the current load call, and modules loaded by a later load call
  - output link and start information to the user
  - support debugging and diagnostic aids (AID commands).
- External symbol dictionary containing references
  (operand LOAD-INFORMATION=*REFERENCES)
  An external symbol dictionary containing the resolved references of all modules in the load unit is loaded.
  References are external references (EXTRNs), V-type constants, weak external references (WXTRNs) and external dummy sections (XDSEC-Rs).DBL needs this information to unlink load units after loading and to perform delayed processing of external references.
- External symbol dictionary for constructing the DBL map
  (operand LOAD-INFORMATION=*MAP)
  Only an external symbol dictionary required for constructing the DBL map is temporarily loaded. The ESD is unloaded as soon as the DBL map is complete.
- LSD information (operand TEST-OPTIONS=*AID)
  LSD (list for symbolic debugging) information is loaded. This information is required by the debugging and diagnostic aids (AID commands). LSD information is useful only when the load unit contains program definitions. The LSD information cannot be loaded if the value specified for the LOAD-INFORMATION operand is *NONE or *MAP.

2.1.17 DBL map

A DBL map containing information about the structure and contents of the loaded load unit is output when the PROGRAM-MAP operand is included in the load call with a command or the MAP operand in the BIND macro. Output of a DBL map is possible only in the ADVANCED operating mode (see "Migration from the dynamic linking loader DLL to DBL"). The SYSOUT or SYSLST system file, or both, may be selected as the output destination. The BIND macro also offers the option of output to a user-defined data area.

The following table provides an overview of the information output for a load unit and the loaded modules.

Information		**Object**								
		A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Object name		X	X	X	X	X	X	X	X	X
Name of the link context		X								
Load information	1)	X								
Type of LLM / PAM-LLM	2)		X	X						
Type of OM	3)				X	X				
Object address							X	X	X	
Object size							X		X	X
Linkage method	4)		X	X		X				
File link name	5)		X	X		X				
Library/ file name	6)		X	X		X				
Element name			X							
Element version			X							

A Load unit

B LLM

C PAM-LLM

D OM in LLM

E OM

F CSECT

G ENTRY

H COMMON

I XDSEC-D

Footnotes

1. The following load information is output:
   - Value of the LOAD-INFORMATION operand
   - Program version of the load unit. If no program version was specified in the load call, DBL shows the PLAM version of the loaded library member (~ stands for the highest PLAM version)
   - Load time
   - Value of the TEST-OPTIONS operand
   - Start address of the load unit (load unit starting point)
   - Addressing mode (AMODE):
     24, 31 or 32
   - as the code type for HSI=:

/7500	The symbol from the load call indicates /390 code
/4000	The symbol from the load call indicates RISC(MIPS) code
/SP04	The symbol from the load call indicates SPARC code
/X86	The symbol from the load call indicates X86 code

2. The type of LLM/PAM-LLM (see the manual “BINDER” [1]) is output as follows:

STANDARD	The LLM consists of a single slice.
BY-USER	The LLM consists of user-defined slices.
BY-ATTR	The LLM consists of slices formed by attributes.

3. The type of OM is output as:

STANDARD	The OM was generated by a compiler.
PRELINK	The OM was generated as a prelinked module in a TSOSLNK linkage run.

4. The bind (linkage) method is output as:

EXPLICIT	The module was loaded as primary input through an explicit specification in the load call.
AUTOLINK	The module was loaded as secondary input by means of the autolink function in order to resolve external references.
INCLUDE	The module was loaded as secondary input as a result of INCLUDE statements inserted in already loaded modules. This is only possible for object modules (OMs).

PUBLIC	An LLM consists of slices formed according to the PUBLIC attribute (see the “BINDER” manual [1]). When the LLM was loaded, the slices were loaded with the PUBLIC attribute into class 6 memory.

5. The file link name of the library in which the LLM/OM was found is output. If a user or system Tasklib is concerned, USERTSKL or SYSTTSKL is output. ECERDLOD is output for PAM-LLMs.

Example of a DBL map (extract)

```
############################## A D B L   M A P ##############################
#                                                                            #
# LOAD UNIT: %UNIT. . . . . . . . . . . . . LOAD INFO  -DEF                  #
# VERSION  : V02.7A40 . . . . . . . . . . . LOAD TIME  -2009-01-30 09:37:27 #
# CONTEXT  : LOCAL#DEFAULT. . . . . . . . . TEST OPTION-NONE                 #
#                                                                            #
#  LLM     : BLSSYS . . . . . . . . . . . . STANDARD / EXPLICIT  BLSLIB     #
#  ELE-LIB : :2OSH:$TSOS.SYSLNK.BLSSERV.027                                  #
#  ELE-NAME: BLSSYS                                                          #
#  ELE-VERS: V02.7A40                                                        #
#   OM     : PBSMAIN. . . . . . . . . . . . STANDARD                        #
#    CSECT : PBSMAIN. . . . . . . . . . . . @-       0 L-     8C0           #
#     ENTRY: NLKRES28 . . . . . . . . . . . @-      18                       #
#     ENTRY: NLKRES45 . . . . . . . . . . . @-      28                       #
#     ENTRY: PBSINI . . . . . . . . . . . . @-      38                       #
#     ENTRY: PBSGOV . . . . . . . . . . . . @-      48                       #
#    CSECT : PBSMAIN@ . . . . . . . . . . . @-     8C0 L-     E00           #
#    CSECT : PBSSSID. . . . . . . . . . . . @-    16C0 L-      80           #
#    CSECT : PBSINIV. . . . . . . . . . . . @-    1740 L-      40           #
#    CSECT : PBSOSDV. . . . . . . . . . . . @-    1780 L-      40           #
#    CSECT : PBSSXTP. . . . . . . . . . . . @-    17C0 L-      40           #
#    CSECT : PBSHWPS. . . . . . . . . . . . @-    1800 L-      40           #
#   OM     : PBSDM. . . . . . . . . . . . . STANDARD                        #
#    CSECT : PBSDM. . . . . . . . . . . . . @-    1840 L-     940           #
#     ENTRY: PBDMUCD. . . . . . . . . . . . @-    1858                      #
#    CSECT : PBSDM@@@ . . . . . . . . . . . @-    2180 L-     280           #
#   OM     : PBSSYSA. . . . . . . . . . . . STANDARD                        #
#    CSECT : PBSSYSA. . . . . . . . . . . . @-    2400 L-     940           #
#     ENTRY: PBSANCH. . . . . . . . . . . . @-    2418                      #
#     ENTRY: PBSTIME. . . . . . . . . . . . @-    2428                      #
#     ENTRY: PBSTERM. . . . . . . . . . . . @-    2438                      #
#     ENTRY: PBSMEMO. . . . . . . . . . . . @-    2448                      #
#     ENTRY: PBSWA. . . . . . . . . . . . . @-    2458                      #
#     ENTRY: PBSSINF. . . . . . . . . . . . @-    2468                      #
#     ENTRY: PBSTINF. . . . . . . . . . . . @-    2478                      #
#     ENTRY: PBSMSG . . . . . . . . . . . . @-    2488                      #
#     ENTRY: PBSBLSP. . . . . . . . . . . . @-    2498                      #
#    CSECT : PBSSYSA@ . . . . . . . . . . . @-    2D40 L-    2FC0           #
#   OM     : PBSBUFD. . . . . . . . . . . . STANDARD                        #
#    CSECT : PBSBUFD. . . . . . . . . . . . @-    5D00 L-    2500           #
#     ENTRY: PBSBDSP. . . . . . . . . . . . @-    5DE0                      #
#     ENTRY: PBSBINI. . . . . . . . . . . . @-    5D00                      #
#   OM     : PBSBUF . . . . . . . . . . . . STANDARD                        #
#    CSECT : PBSBUF . . . . . . . . . . . . @-    8200 L-    2040           #
#     ENTRY: SVPTBUF. . . . . . . . . . . . @-    8208                      #
#     ENTRY: SVSVC183 . . . . . . . . . . . @-    8250                      #
#     ENTRY: SVSVC117 . . . . . . . . . . . @-    8340                      #
#     ENTRY: SVSVC111 . . . . . . . . . . . @-    8588                      #
#     ENTRY: SVSVC110 . . . . . . . . . . . @-    83E8                      #
#     ENTRY: SVSVC109 . . . . . . . . . . . @-    8500                      #
#     ENTRY: SVSVC106 . . . . . . . . . . . @-    8478                      #
#     ENTRY: $SVBDAS. . . . . . . . . . . . @-    8A50                      #
#     ENTRY: $SVBDSP. . . . . . . . . . . . @-    8B20                      #
#     ENTRY: $SVUNAS. . . . . . . . . . . . @-    8A50                      #
#     ENTRY: $SVUNSP. . . . . . . . . . . . @-    8B20                      #
#     ENTRY: $SVVIAS. . . . . . . . . . . . @-    8A50                      #
#     ENTRY: $SVVISP. . . . . . . . . . . . @-    8B20                      #
.
.
.
# LOAD UNIT STARTING POINT. . . . . . . . . @-       0 AMODE-24 HSI-/7500   #
#                                              MMODE-TU-4K                   #
#                                                                            #
######################## E N D   O F   A D B L   M A P ########################
```

Layout of the DBL map in a user-defined data area

`BIND ...,USRMAPI=...,USRMAP@=...,USRMAPL=...` (where `USRMAPIîNONE`) enables a DBL map to be output to a user-defined data area. This map is structured in a similar way to the list output to SYSOUT/SYSLST. It has the following layout:

- Map header for identifying the area
  This contains the return code X'08010129' if the length of the data area is not sufficient to contain the required information.
  The layout of the list header depends on the INTVERS specification in the BIND macro: INTVERS=SRV005 generates a list header version 1, INTVERS=SRV006 generates a list header version 2. This shows whether the library name, element name and element version are contained in the information output.
- Total length of the information output, including map header
- Data records with the following layout:
  - Record ID
  - Length of the information contained in the record
  - The information itself

In the case of INTVERS=SRV005 the user-defined data area must be at least 248 bytes long, and in the case of INTVERS=SRV006 at least 336 bytes long to accommodate the minimum information.

If the length of the data area is not sufficient to contain the required information, output is aborted.

If the length of the data area is sufficient, the output is terminated by an end record.

The layout of the data area can be generated with
`BIND MF=D,XPAND=USRMAP,INTVERS=SRVxxx` (xxx >= 005).

2.1.18 Shareable programs (shared code)

Each module executed by a user is loaded into either the task-local class 6 memory or the memory for shareable programs (shared code). The modules in the task-local class 6 memory can be executed only by the task which caused them to be loaded there. These modules form the so-called private part of the program.

The memory for shared code is a memory area in which one and the same copy of the module to be executed can be executed simultaneously by several tasks. These shareable modules form the public part of a program and must be programmed as reentrant code. Such a program must dynamically request the data areas in which variable user-owned data is to be stored, and must create these areas in the class 6 memory of the related user. The user can either directly execute the shared code (with START-EXECUTABLE-PROGRAM or START-PROGRAM) or access it via a private program. The external references in private programs can be resolved automatically by program definitions (CSECTs, ENTRYs) in the shared code.

Advantages of shareable programs

- Since a module is loaded only once, i.e. when it is first referenced, the loading time is saved for all subsequent calls to the module during the same session.
- Demands on main memory and paging memory are reduced because all tasks can access a single copy of the object module in the shared code memory, i.e. there is no need for each task to have a separate copy in its own class 6 memory (see figure 5 for an example of system shared code).
- The paging rate is reduced because only *one* copy resides in the paging area. Also, read-only pages do not have to be written out to paging memory when they are no longer needed in main memory, i.e. no updating of paging memory is necessary for them.

Program shareability is therefore worthwhile for larger programs which are to reside in memory for some time (in interactive and transaction processing mode) and possibly be used concurrently by a number of users.

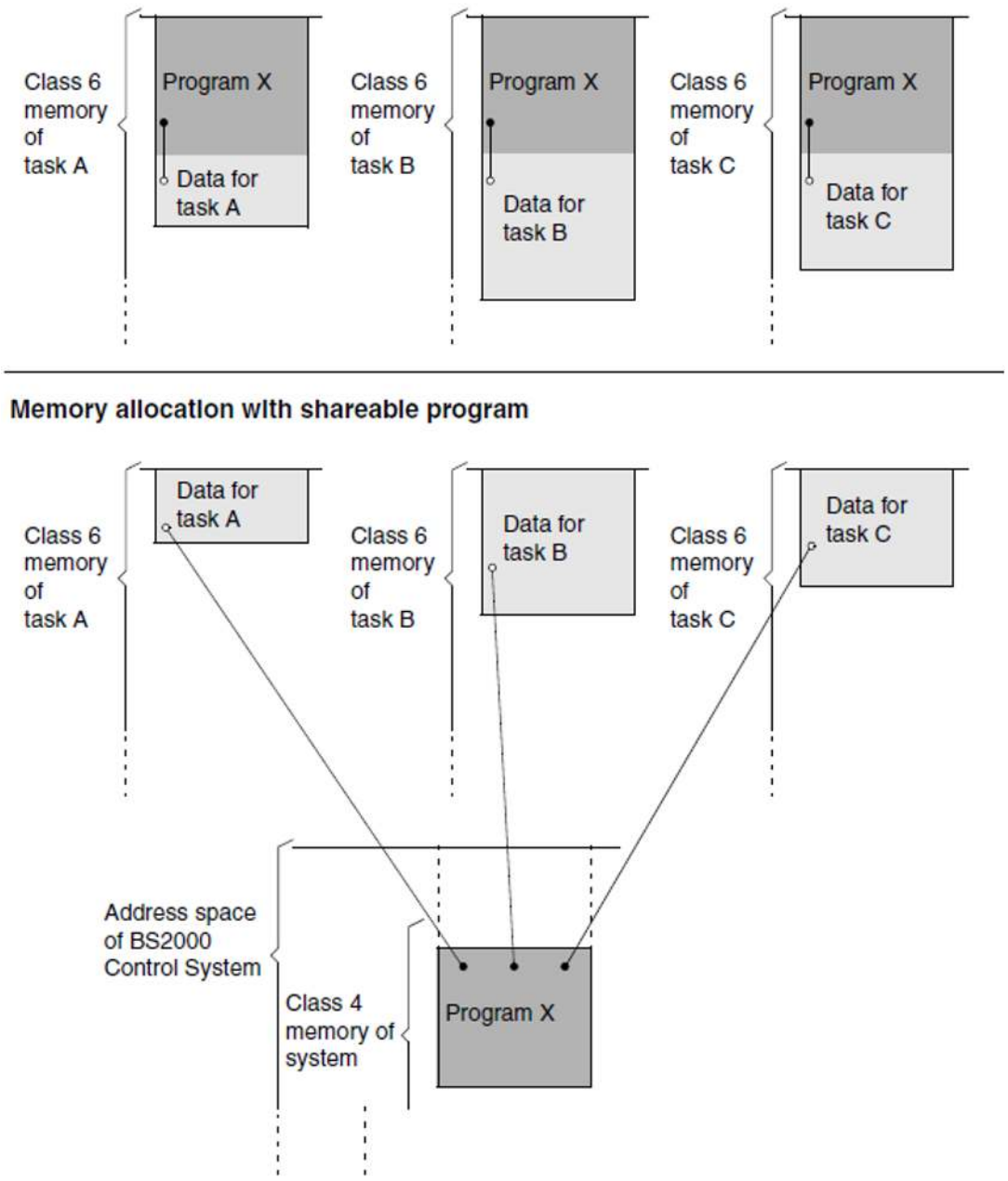


Figure 5: Effects of program shareability

DBL offers various facilities for loading and managing shared code.

2.1.19 System shared code

The system shared code is loaded into the system address space (class 3/4 memory or privileged class 5 memory), and all tasks or users can access it. DBL ensures that this shared code is always write-protected, regardless of the value of the READ-ONLY attribute.

This memory area is controlled by the subsystem management (DSSM), since the shared code was loaded as nonprivileged subsystems (see the “Subsystem Management” manual [10]). For each subsystem, a list of visible symbols (which are to be known outside the subsystem) must be defined in the DSSM. Only these symbols, which are called “connectable entries”, may be specified in a START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM command or in the BIND macro and are available for the resolution of external references.

2.1.20 User shared code

Shared code which is to be managed by a user without special privileges is stored in common memory pools, which are created in class 6 memory. The modules loaded there can be shared by all tasks which are connected to the memory pool.

The shared code can be loaded into the common memory pool with the ASHARE macro. Each task which is connected to the common memory pool can:

- load a shareable program (consisting of a set of modules) into the memory pool
- unload a shareable program
- request information about the loaded modules.

The name of the program and all visible symbols in the user shared code may be specified in the operand SYMBOL of the START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM commands and in the BIND macro, and may also be used for resolution of external references. If a task which is not yet connected to the common memory pool refers to program definitions in the shared code in this memory pool,
DBL automatically executes the ENAMP macro (ENAble Memory Pool, see the “Executive Macros” manual [7]).

The BIND macro (MPID parameter) can also be used to load a program into the common memory pool. However, programs loaded in this manner can be accessed only by the task which initiated loading into the memory pool. This means that this task is responsible for management of the memory pool. Information about a program loaded in this manner (e.g. the output from the VSVI1 macro) is also available only to the task which loaded the program.

In the following text, it is assumed that the shared code was loaded into the common memory pool with the ASHARE macro.

> ¡ A program can be loaded into one and the same memory pool only once with the BIND macro or with the ASHARE macro.

2.1.21 LLMs with PUBLIC slices

When an LLM is generated by BINDER, the user can specify whether slices are to be formed on the basis of the attribute PUBLIC. If so, the LLM is split into two parts (slices), each of which has the form of a simple LLM. The slice containing the CSECTs with the attribute PUBLIC=YES can be loaded as shared code; the other slice, which contains all CSECTs with the attribute PUBLIC=NO, can be loaded only in the task-local user memory and forms the private part of the LLM.

DBL loads only the PUBLIC part of the LLM if the LLM is loaded with the DSSM as a nonprivileged subsystem or with the ASHARE macro.

If the LLM is loaded with the START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM command or with the BIND macro, the private part of the LLM is loaded first. If this private part contains external references to the PUBLIC part, two possible situations exist:

1. The PUBLIC part of the LLM has already been loaded into the shared code memory. In this case, the external references in the private part are resolved by the shared code.
2. The PUBLIC part is not yet in the shared code memory. In this case, the PUBLIC part is loaded as private code into the task-local user memory. The same thing happens if the user specified SHARE[-SCOPE]=NONE in the load call because the PUBLIC part is not to be used as shared code.

When loading the private part of an LLM which contains external references to the PUBLIC part, DBL checks whether both parts actually belong to the same LLM. If this is not the case, the PUBLIC part of the LLM is loaded into the task-local user memory as private code.

If the PUBLIC part of the LLM is to be loaded via DSSM, the user must specify the list of SUBSYSTEM-ENTRIES which are to be visible outside the PUBLIC part of the LLM. This list must be specified in the statement START-LLM-CREATION or MODIFY-LLM-ATTRIBUTES during the BINDER run. This facility is available as of BINDER V1.1A.

2.2 Context concept

The term **context**, as used below, has three different meanings.

A context may be:

- a set of objects with a logical structure
- an environment for linking and loading
- an environment for unloading and unlinking.

The use of contexts by DBL 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. In each individual context external references are resolved separately. Each sub-application in a context can therefore be loaded and run as an autonomous entity. In this way it is possible, for example, to load and start the individual modules of a runtime system in separate contexts.
- Parts of an application that belong to a context can be unloaded with a *single* call.
- Identically named symbols in different contexts do not provoke name conflicts because each context maintains its own symbol table.

The various context concepts are explained in more detail below.

2.2.1 Context as a set of objects

A context as a set of objects has a logical structure. It is structured hierarchically as follows (see figure 6):

1. Objects at the lowest level are **program definitions**. They include the following objects:
   - control sections (CSECTs)
   - entry points (ENTRYs)
   - COMMON sections
   - dummy sections (DSECTs)
   - external dummy sections (XDSEC-Ds)
2. Objects at the next higher level are **modules**. They contain the program definitions. Modules can be object modules (OMs) or link and load modules (LLMs).
3. Objects at the next higher level are referred to as **load units**. A load unit contains all the modules that are linked and loaded with *one* load call.
4. The context itself consists of one or more load units. It is the highest-level object.

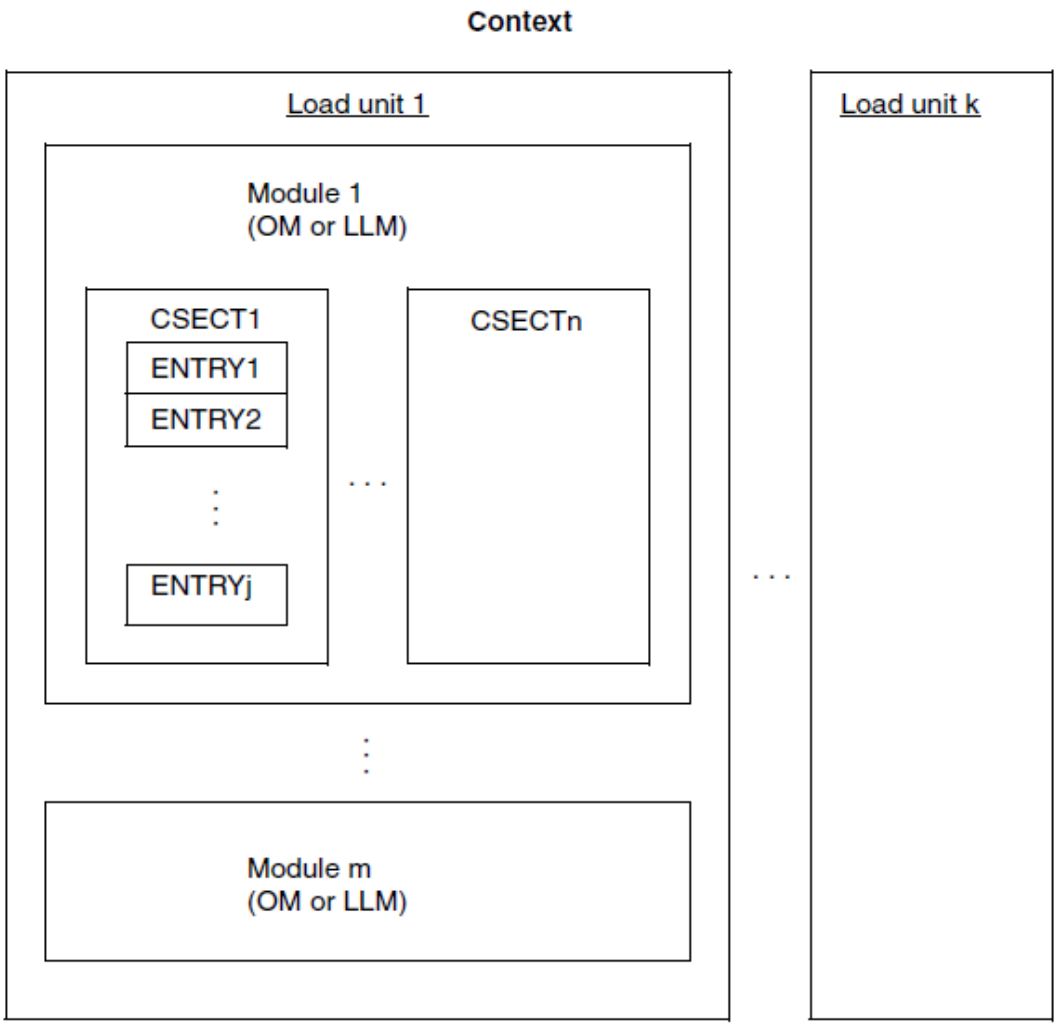


Figure 6: Logical structure of a context as a set of objects

2.2.2 Context as a linking and loading environment

This context is a main memory area into which the modules of a load unit are loaded. The area is set up dynamically by DBL. In other words it is not possible for the user to reserve a fixed space for it. The area can be completely contained within a memory pool or spread over several pools. In the latter case, however, it may only be located within a single memory class. For the modules loaded in this main memory area, there is a symbol table in which an address is assigned to each symbol.

The context for linking and loading can be used as a link context or reference context:

Link context

The modules of a load unit are loaded into the link context. All symbols of a module which is loaded into this context are entered in the context symbol table. The link context is used by the autolink function of DBL for resolving external references in the loaded modules (see "Resolving external references").

It is also used to buffer unresolved external references if this is specified in the load call (operand UNRESOLVED-EXTRNS=*DELAY). If a new load unit is loaded into the link context, DBL tries to resolve the buffered external references with CSECTs and ENTRYs of the new load unit at the end of the loading operation. This process is repeated each time another load unit is loaded, for as long as the context exists.

The user defines the name of the link context in the BIND macro (LNKCTX@ or LNKCTX operand). If no name is specified, DBL selects LOCAL#DEFAULT as the default name. When DBL is invoked with a LOAD- or START-EXECUTABLE-PROGRAM (or LOAD- or START-PROGRAM) command, it defines LOCAL#DEFAULT as the default name.

Reference context

The reference context is used to resolve external references that are not found in the link context. A reference context originates as a link context in another loading operation. Multiple reference contexts (up to 200) may be present simultaneously. The user defines the names and number of possible reference contexts in the BIND macro (REFCTX@ and REFCTX# operands).

2.2.3 Context as an unloading and unlinking environment

A context is the largest entity that can be unloaded (see "Unloading and unlinking objects"). Lower-level objects that can be unloaded are load units and modules (LLMs or OMs).

2.2.4 Properties of a context

Every context has two properties: a scope and an access privilege.

The **scope** of the context determines the class of memory in which the context is to be physically located. A distinction is made between the following scopes:

- USER scope
  The context is located in the user address space (class 6 memory). The context name begins with a letter. The number of USER contexts is limited to not more than 201. For shared code in common memory pools, the scope is also USER. The name of this context begins with “#”. The number of USER pool contexts in a memory pool is limited to not more than 15.
- SYSTEM scope
  The context is located in the system address space (class 4 or class 3 memory). The context name begins with a dollar sign ($).

The link context and the reference context for nonprivileged users must always be in the USER scope.

The **access privilege** determines which users are authorized to access the contents of the context. Contexts may be privileged or nonprivileged:

- Privileged context
  Only privileged users (such as the system administration) may access the contents of the context.
- Nonprivileged context
  Any user may access the contents of the context. Access to the context of a memory pool is permitted only for the users connected to this memory pool.

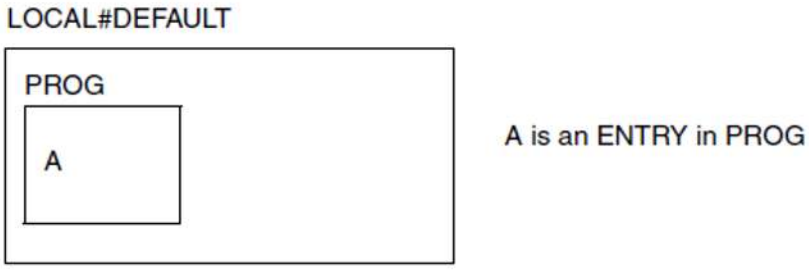
2.2.5 Example of the use of contexts

The following example shows how contexts can be used for avoiding name conflicts and for controlled resolution of external references.

1. A program stored as an element with the name PROG in a program library is called. DBL generates the default link concept LOCAL#DEFAULT for resolution of external references.

   ```
   /START-EXECUTABLE-PROGRAM FROM-FILE=(LIBRARY=...,ELEMENT-OR-SYMBOL=PROG)
   ```

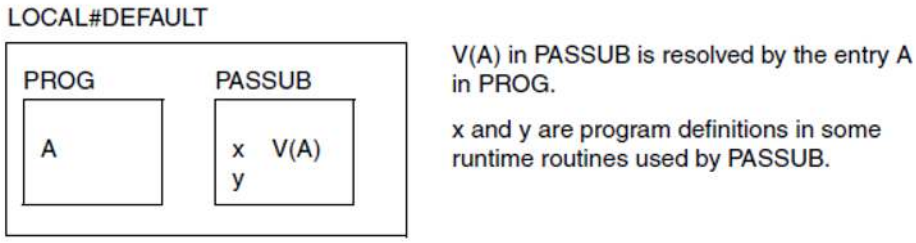
   Result

2. During program execution, PROG calls the BIND macro in order to activate a PASCAL subroutine.

   ```
   BIND SYMBOL=PASSUB,LIBNAM=...
   ```

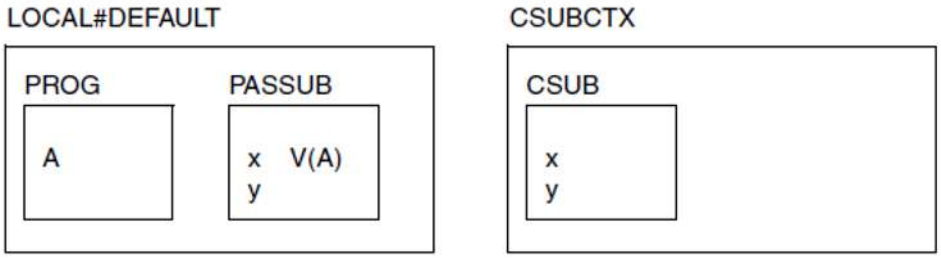
   Result

3. Subsequently, PROG calls the subroutine CSUB, which is written in a different programming language (in C) than that used for PASSUB but also uses runtime routines called x and y. In order to avoid name conflicts, the BIND macro must be called as follows:

   ```
   BIND SYMBOL=CSUB,LIBNAM=...,...,LNCTX=CSUBCTX
   ```

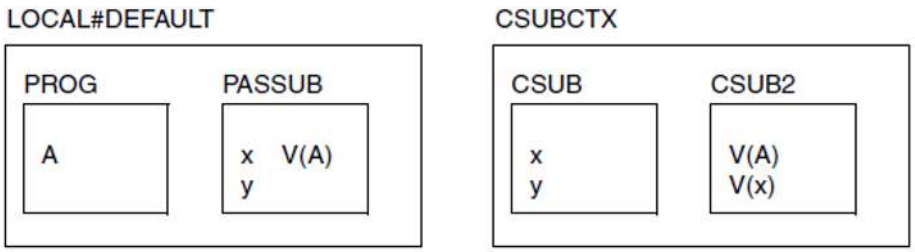
   Result

4. A further subroutine, which is also written in C and which, among other things, contains an external reference to ENTRY A in the main program PROG is called. External reference V(A) is to be resolved with ENTRY A in PROG, while other external references are to be resolved by the runtime routines written in C. For this reason, PROG must call the subroutine CSUB2 as follows:

   ```
   BIND SYMBOL=CSUB2,LIBNAM=...,...,LNKCTX=CSUBCTX,REFCTX=LOCAL#DEFAULT
   ```

   Result



   V(A) is resolved by A in PROG.
   V(x) is resolved by x in CSUB.

2.3 Additional functions

In addition to linking and loading, DBL performs the following functions:

- unloading and unlinking of objects
- output of link and load information
- creating a separate symbol table
- processing REP files

2.3.1 Unloading and unlinking objects

When a program is terminated with the TERM macro or the CANCEL-PROGRAM or EXIT-JOB (LOGOFF) command, all of the user memory space previously occupied by DBL or the static loader ELDE is released.

The UNBIND macro can be used during the program run to release the memory space occupied by objects that are no longer required. The object may be a context, a load unit, an LLM or an object module (OM). Object modules and LLMs can be unloaded only as entities. The objects can also be specified via a path, for example:

- a module in a specific context
- a module in a specific load unit with a defined version
- a module in a specific load unit with a defined version in a specific context
- a load unit with a defined version in a specific context.

Modules which are part of a list name unit (see "Management of list name units") cannot be unloaded individually, but only with the entire list name unit.

In each case, only the first object matching the path which is found is unloaded. Occupied memory space is always released in page-sized units (1 page = 4 Kbytes). The memory space is returned to memory management only if no other modules require space on the same page. Otherwise DBL notes the free areas and uses them at the next opportunity.

In addition to unloading objects, it is also possible to unlink external references to control sections (CSECTs) and entry points (ENTRYs) in these objects (parameter UNLINK=YES).

Unlinking means that resolved external references to CSECTs and ENTRYs in the unloaded objects are released. Resolved XDSECs are an exception. The external references in the objects are then equivalent to unresolved external references at the time of linking and loading. They are given the value specified for unresolved external references in the load call (ERROR-EXIT operand) and, if this is specified in the load call, are treated as external references which are to be resolved at a later time.

Unlinking is performed only within the context in which the module is being unloaded. eferences in other contexts to the object being unloaded are not unlinked. Unlinking is performed only if the external references belong to a load unit for which an external symbol dictionary with references was loaded in the load call (operand LOAD-INFORMATION= *REFERENCES).

2.3.2 Outputting link and load information

The user can request link and load information about the linked load units and their contexts using the VSVI1 macro. This information can be requested for one or more specified contexts or for all contexts. Users without special privileges may access only nonprivileged contexts. Information about shared code in common memory pools is available only to users who have connected themselves to the common memory pool before calling the VSVI1 macro.

The following information can be requested:

- a list of context names (SELECT=CTXLIST)
- the scope of the code loaded in a context, and the scope of the associated link and load information (SELECT=CTXSIZE)
- a list containing names, load addresses, lengths, types, attributes and contexts of all CSECTs, ENTRYs and COMMONs (SELECT=ALLLIST)
- a list containing names, load addresses, lengths, types, attributes and contexts of all CSECTs and COMMONs (SELECT=MODLIST)
- a record containing name, load address, length, type, attributes and context of a *single* control section (CSECT), ENTRY or COMMON (SELECT=BYNAME)
- a record containing name, load address, length, type, attributes and context of a control section (CSECT) or COMMON specified by means of its address (SELECT=BYADDR).

The various items of information (name, load address, length, attribute, type and context) can be selected independently of one another. It is also possible to request merely the length of the desired output information.

The PINF macro returns global information about programs which have been loaded with the LOAD- or START-EXECUTABLE-PROGRAM (or LOAD- and START-PROGRAM) command.

The following information can be requested:

- the internal program name (SELECT=INTNAME)
- the internal program version (SELECT=INTVERS)
- the creation date of the program (SELECT=INTDATE)
- the copyright name of the program (SELECT=COPRIGHT)
- the name of the library in which the program was stored (SELECT=FILENAME)
- the element name of the program in the library (SELECT=ELEMNAME)
- the element version (SELECT=ELEMVERS)
- the element type (SELECT=ELEMTYPE)
- the name specified in the load call (in LOAD- or START-EXECUTABLE-PROGRAM or LOAD- and START-PROGRAM) (SELECT=SPECNAME)
- an indicator which shows whether the program was loaded by the static loader ELDE or whether it was the first module of a load unit and was loaded by DBL (SELECT=LOADTYPE).

2.3.3 Creating a separate symbol table

The ETABLE and ETABIT macros can be used to create a separate context-specific symbol table for program definitions in the loaded program. This table serves as a means of passing symbolic information to DBL and thus enables the exchange of information between different program sections.

> The ETABLE macro has replaced the earlier TABLE macro. Note, however, that the handling of name conflicts for BIND and ETABLE is no longer the same as for the old BIND and TABLE macros. The handling of name conflicts for ETABLE is described in the „Executive Macros“ manual [7].

2.3.4 Processing REP files

REP files may be specified in all load calls with the LOAD- or START-EXECUTABLE-PROGRAM (or LOAD- and START-PROGRAM) commands and with the ASHARE and BIND macros.

A REP file is always associated with a load unit. The REP records contained in it can be used for bytewise corrections to the individual modules of a load unit by the same method as when loading the BS2000 Control System (see the “Introductory Guide to Systems Support“ [9].
The REP records may be applied to all modules in the link context or be restricted to just the modules of the load unit (REPSCOP=CONTEXT or UNIT). Masked symbols may also be included for REP processing in the link context.

When the load operation has been completed, DBL opens the REP file associated with the load unit, reads in the REP records, and then applies them to the modules of the load unit or to all modules in the link context, taking the NOREF file, if any, into account. A NOREF file contains a list of symbols (CSECTs and ENTRYs) that are not relevant for REP processing (see "NOREF files").

Corrections that affect code in a system context are controlled by the BLS Lock Manager. The system context involved is protected against any other write access during the entire link/load process (including REP processing).

Format of REP records and REP files

REP records must be supplied in the same format that is used for REPs of the BS2000 Control System. This also applies to relocatable REPs. A detailed description of the format of REP records and REP files can be found in the “Introductory Guide to Systems Support” [9].

Every REP record is checked for the correct format before it is processed. This includes, among other things, the data contained in it, the parity byte, the module version, and the module name. The class identifier (column 70) has no significance for load units and is therefore not taken into account.

Any invalid or unrecognized REP records are logged on SYSOUT, and processing continues with the next REP record.

If the REP file specified in the load call does not exist, REP processing is aborted with error message BLS0977 /BLP0977, and the load unit is loaded without REP corrections. In the case of other DMS errors on accessing the REP file, DBL displays message BLS0998/ BLP0998 and offers a second access attempt. If this attempt also fails, the load unit is loaded without REP corrections.

REP files for subsystems

A REP file that is associated with a subsystem must comply with the following naming convention:

`SYSREP.<subsystem-name>.<subsystem-version>`

The REP file should be shareable, especially if the subsystem has not been preloaded, but is loaded instead at the first call.

Return information and error flags

The following BLS messages are output when processing REP files:

BLS0990/BLP0990	on successfully opening the REP file
BLS0980/BLP0980	for each comment in the REP file

BLS0989/BLP0989, BLS0991/BLP0991, BLS0992/BLP0992, BLS0993/BLP0993, BLS0998/BLP0998	if errors occur during REP processing

NOREF files

A NOREF file is associated with a REP file and contains a list of CSECTs and ENTRYs that need not be taken into account during REP processing. In other words, if DBL cannot find a CSECT or an ENTRY for which there is a REP record, DBL will search for it in the NOREF file. If the symbol in question is listed in the NOREF file, DBL will skip that REP record without an error message.

This enables a REP file to be applied to:

- software products consisting of multiple subsystems or load units
- subsystems or programs consisting of multiple load units.

> The character “S” (for selectable unit) in column 69 of a REP record has the same effect as a the corresponding entry in the NOREF file.

Format of NOREF files

A NOREF file is a SAM file of variable length. It consists of 8-byte records (one record per symbol name), which are delimited by blanks. The first record of the NOREF file contains the number of subsequent records in the first four bytes in hexadecimal notation.

Naming convention for NOREF files

DBL can process any NOREF file that complies with the naming conventions for BS2000 REP files:

- If the REP file name specified in the load call includes the name component “SYSREP”, DBL replaces that name component with “SYSNRF” and searches for a NOREF file under that name.
- Otherwise, DBL searches for a NOREF file with the same name as the main library, but with the additional suffix “.NOREF”.

DBL does not issue any message if no NOREF file is found.

On opening the NOREF file successfully, DBL displays message BLS0995/BLP0995. Message BLS0996/BLP0996 or BLS0997/BLP0997 is output if an error occurs during NOREF processing.

2.4 Execution of the dynamic binder loader

- Calling DBL
- Message handling

2.4.1 Calling DBL

The user activates DBL by means of commands or macros, and controls its execution with the operands of these commands or macros.

The following **commands** invoke DBL:

- The commands START-EXECUTABLE-PROGRAM or LOAD-EXECUTABLE-PROGRAM if no load modules (load module file or type C library element) are loaded.The START-EXECUTABLE-PROGRAM command binds modules to form a load unit, loads this into main memory and then starts it. If users want to load the load unit without starting it then they can use the LOAD-EXECUTABLE-PROGRAM command.
- The START-PROGRAM or LOAD-PROGRAM command if the *MODULE operand was specified.
  These commands are now only supported for reasons of compatibility.
  The START-PROGRAM command links modules together into a load unit, loads this into main memory and starts it. If the user merely wishes to load the load unit without starting it, the LOAD-PROGRAM command can be used instead.
- The CANCEL-PROGRAM command
  This command terminates the program run and releases all of the user memory space occupied by DBL and the static loader ELDE.
- The MODIFY-DBL-DEFAULTS, RESET-DBL-DEFAULTS and SHOW-DBL-DEFAULTS commands set and display default values for the DBL run.
- The SELECT-PROGRAM-VERSION command
  This command defines which version of a program is used in cases where DBL can access multiple program versions.

The following **macros** invoke DBL:

- The BIND macro
  This links a further load unit into the executing program.
- The UNBIND macro
  UNBIND is used during the program run to release memory space occupied by objects that are no longer required. The object may be a context, a load unit, an LLM or an object module.
- The ASHARE macro
  ASHARE links and loads shared code which the user wishes to place as shareable code in the common memory pool.
- The DSHARE macro
  DSHARE unloads from common memory pools any code loaded with the ASHARE macro.
- The LDSLICE macro
  LDSLICE loads a slice defined by the user in an LLM into main memory.
- The VSVI1 macro
  VSVI1 provides the user with link and load information concerning the linked load units and their contexts.
- The PINF macro
  PINF returns global information on loaded programs to the user.
- The ILEMGT and ILEMIT macros
  ILEMGT and ILEMIT are used to create and manage ILE lists.
- The ETABLE and ETABIT macros
  ETABLE and ETABIT are used to create a user-defined symbol table in the context.

- The GETPRGV macro
  GETPRGV returns the currently selected program version to the user.
- The SELPRGV macro
  SELPRGV can be used to select a specific program version.

2.4.2 Message handling

For message handling by DBL, the messages are arranged into certain **message classes**, as follows:

INFORMATION	Informational message
WARNING	Warning message
ERROR	Error message

Each message class has a particular **weighting**. The INFORMATION message class has the lowest weighting, and the ERROR message class the highest.

By including the MESSAGE-CONTROL operand in the load call it is possible to define which classes of messages are to be output. This operand determines the lowest message class, i.e. messages at and above this level will be output. The table below shows how the MESSAGE-CONTROL operand controls message output.

	MESSAGE-CONTROL			
Message class	*INFORMATION	*WARNING	*ERROR	*NONE
INFORMATION	yes	no	no	no
WARNING	yes	yes	no	no
ERROR	yes	yes	yes	no

Some of the information messages are suppressed if job switch 4 is set. The HELP-MSG-INFORMATION command provides you with an explanation of the meaning of the messages output by DBL.

Depending on the value of the system parameter EACTETYP (see the “Introductory Guide to Systems Support“ manual [4]), some BLS messages are output to SYSOUT and/or the operator console.

Message codes

As with all BS2000 messages, the message codes of the BLS messages are seven characters long, the first three letters standing for the BS2000 message class and the last four characters specifying the message number. The BS2000 message class for BLS messages depends on whether or not the BLSSERV subsystem is already loaded. From the start of a session to the point when the BLSSERV subsystem is loaded, only a subset of the BLS functionality is available, and the BS2000 message class of the messages is “BLP”. Subsequently the complete BLS functionality is available, and the BS2000 message class of the messages is “BLS”. For example, each time a REP file is assigned, message BLP0990 or BLS0990 is issued depending on whether the assignment took place before or after the BLSSERV subsystem was loaded. The meaning of the BLSxxxx and BLPxxxx messages with the same number is as a rule identical. However, some inserts or explanations may differ.

2.5 Commands

The following overview contains all commands which call DBL. The detailed description of the commands can be found in the „Commands“ manual [5].

Command	Function
CANCEL-PROGRAM	Unload program
LOAD-EXECUTABLE-PROGRAM	Load program
LOAD-PROGRAM	Load program
MODIFY-DBL-DEFAULTS	Set defaults for DBL calls
RESET-DBL-DEFAULTS	Reset defaults for DBL calls
SELECT-PROGRAM-VERSION	Select program version
SHOW-DBL-DEFAULTS	Show defaults for DBL calls
START-EXECUTABLE-PROGRAM	Load and start program
START-PROGRAM	Load and start program

2.6 Macros

The following overview contains all DBL macros. DBL macros for earlier versions of BS2000 (up to BS2000 V9.5), except for the TABLE macro, are now supported at the object level only. The TABLE macro is described in the appendix. The macros are descibed in detail in the „Executive Macros“ manual [7].

Macro	**Function**
ASHARE	Load shared code into a common memory pool
BIND	Link and load load unit
DSHARE	Unload shared code from common memory pools
ETABIT *)	Create load information table
ETABLE *)	Pass load information (extended TABLE)
GETPRGV *)	Get program version
ILEMGT *)	Manage ILEs (Indirect Linkage Entries)
ILEMIT *)	Create ILE table entry
LDSLICE	Load slice
PINF	Output global information on a loaded program
SELPRGV *)	Select program version
UNBIND	Unload and unlink
VSVI1	Output link and load information
*) These macros need ASSEMBH.	

3 XS support for DBL

This chapter is intended for users wishing to use the address space above 16 Mb. It is assumed that readers are familiar with the basics of XS programming.

3.1 Determining the addresses passed by the user

The addresses passed by the user in the ASHARE, BIND, DSHARE, LDSLICE, PINF, UNBIND and VSVI1 macros are determined by DBL as follows:

- For the INADDR operand of the VSVI1 macro, the address is always specified as a 31-bit address.
- For the other operands in the macros, the address is determined by the addressing mode of the calling program. If AMODE 31 is set, a 31-bit address is specified; if AMODE 24 is set, a 24-bit address.

3.2 Pseudo-RMODE of an object module

In order that the dynamic binder loader DBL can load an object module as a unit it is necessary to specify the general location of the module in the address space (above or below 16 Mb). To this end, a pseudo-RMODE is defined for the module at linkage time. This is determined from the RMODE attributes of the individual CSECTs, as follows:

- The object module is assigned the attribute (pseudo-)RMODE ANY only if all CSECTs contained in it have the RMODE ANY attribute.
- If at least one CSECT has the attribute RMODE 24, the module also is assigned the restricted attribute (pseudo-) RMODE 24.

The AMODE attribute is determined by the control section (CSECT) which contains the entry point of the object module.

3.3 Pseudo-RMODE of an LLM or PAM-LLM

The following details apply for PAM-LLMs and also for the earlier LLMs.

In order that the dynamic binder loader DBL can load an LLM as a unit it is necessary to specify the general location of the LLM in the address space (above or below 16 Mb). To this end, a pseudo-RMODE is defined for the LLM at linkage time. This is determined in one of the following two ways:

1. For an LLM consisting of a single slice (SINGLE) or of user-defined slices (BY-USER), BINDER determines the pseudo-RMODE from the RMODE attributes of the individual CSECTS and possibly also from the attributes of the COMMONs, as follows:
   - The LLM is assigned the attribute (pseudo-)RMODE ANY only if all CSECTs contained in it have the RMODE ANY attribute.
   - If at least one CSECT has the attribute RMODE 24, the LLM also is assigned the restricted attribute (pseudo-)RMODE 24.

   This type of LLM can only be loaded in its entirety either above or below 16 Mb.
2. For an LLM consisting of slices which were formed from attributes (BY-ATTRIBUTES), BINDER determines a separate pseudo-RMODE for each slice, even if the RMODE attribute was not used to form the slices. The pseudo-RMODE for each slice is determined from the RMODE attributes of the individual CSECTS, as described under point 1.

   This type of LLM can be loaded partially above and partially below 16 Mb.

3.4 Determining the load address

The load address and addressing mode of a load unit above 16 Mb are defined by the operand PROGRAM-MODE=24/ANY in the START/LOAD-EXECUTABLE-PROGRAM or START/LOAD-PROGRAM command or the BIND macro.

DBL determines the load address and the addressing mode of the load unit

- from the PROGRAM-MODE operand and
- from the AMODE and RMODE attributes of the CSECTs in the load unit.
- If AMODE-CHECK=*ADVANCED is specified, the AMODE attribute of the load unit is also used to determine the load address and addressing mode of the load unit.

3.4.1 LOAD- and START-EXECUTABLE-PROGRAM (or LOAD- and START-PROGRAM) commands

PROGRAM-MODE=24 (default)

DBL evaluates these operands as follows:

- The load unit is loaded below 16 Mb.
- External references are resolved only with CSECTs or ENTRYs located below 16 Mb.
- 24-bit addressing mode is set.
- Loading of the load unit is aborted with an error message if it contains a CSECT with the AMODE 31 attribute.

PROGRAM-MODE=ANY

Each module of the load unit can be loaded above or below 16 Mb. The load address is dependent on the RMODE attributes of the CSECTs contained in the module.

If the module contains a number of CSECTs, DBL defines a (pseudo-)RMODE that is derived from the RMODE attributes of the individual CSECTs as follows:

- The module is assigned the attribute pseudo-RMODE ANY only if all CSECTs contained in it have the RMODE ANY attribute.
- If at least one CSECT has the attribute RMODE 24, the module also is assigned the attribute pseudo-RMODE 24.
- If AMODE-CHECK=*ADVANCED is specified, the AMODE attribute of the load unit is also used to determine the (pseudo-)RMODE and load address of the load unit.

When AMODE-CHECK=*STD the load address is determined by the (pseudo-)RMODE as follows:

(Pseudo-)RMODE	
24	All load unit modules are loaded below the 16 MB boundary
ANY	All load unit modules are loaded above the 16 MB boundary

When AMODE ANY is set, the addressing mode is determined by the location of the entry point. If this is below 16 Mb, 24-bit addressing mode is set; if it is above 16 Mb, 31-bit addressing mode is set.

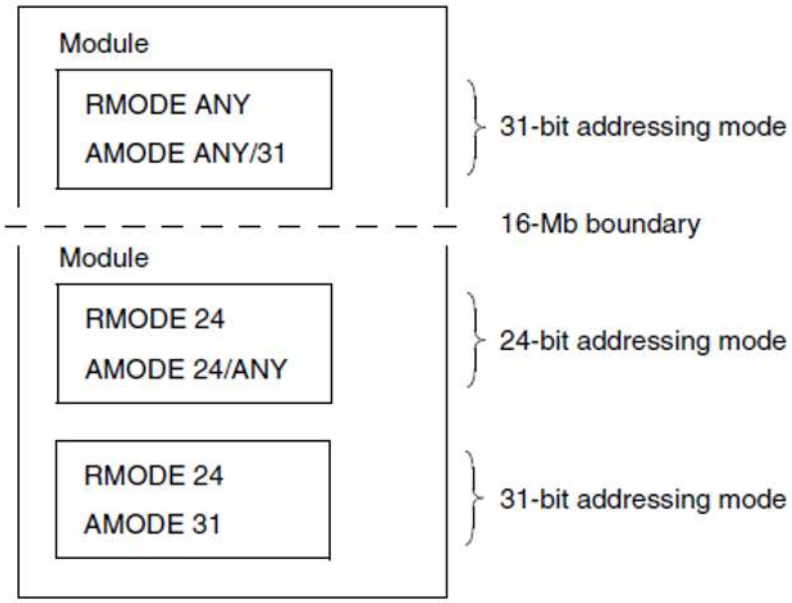


Figure 7: Addressing mode on START-/LOAD-EXECUTABLE-PROGRAM

With AMODE-CHECK=*ADVANCED, the following applies:

If after the first module of the load unit has been loaded 24 was determined as AMODE attribute, all other modules of the load unit are loaded below 16 Mb.

3.4.2 BIND macro

With the BIND macro, DBL determines the load address and the addressing mode from the operands PROGMOD=ANY|24 and BRANCH=NO|YES.

PROGMOD=ANY (default)

Each module of the load unit can be loaded above or below 16 Mb. The load address is dependent on the RMODE attributes of the CSECTs contained in the module and on the program addressing mode at the time of the BIND call.

If AMODE-CHECK=*ADVANCED is specified, the AMODE attribute of the load unit is also used to determine the load address of the load unit's modules.

If the module contains a number of CSECTs, DBL defines a (pseudo-)RMODE based on the RMODE attributes of the individual CSECTs, as follows:

- The object module is given the attribute pseudo-RMODE ANY only if all the CSECTs contained in it have the RMODE ANY attribute.
- If at least one CSECT has the RMODE 24 attribute, the module also is assigned the attribute pseudo-RMODE 24.

When AMODE-CHECK=*STD the load address is determined as follows:

- If the program addressing mode on the BIND call is not 31 then all the load unit modules are loaded below the 16 MB boundary.
- Otherwise the modules are loaded above or below the 16 MB boundary depending on their (pseudo) RMODE.

The DBL thus only loads any module above the 16 MB boundary if the module possesses the attribute RMODE ANY and the 31-bit addressing mode was set in the BIND call.

In all other cases the module is loaded below 16 Mb.

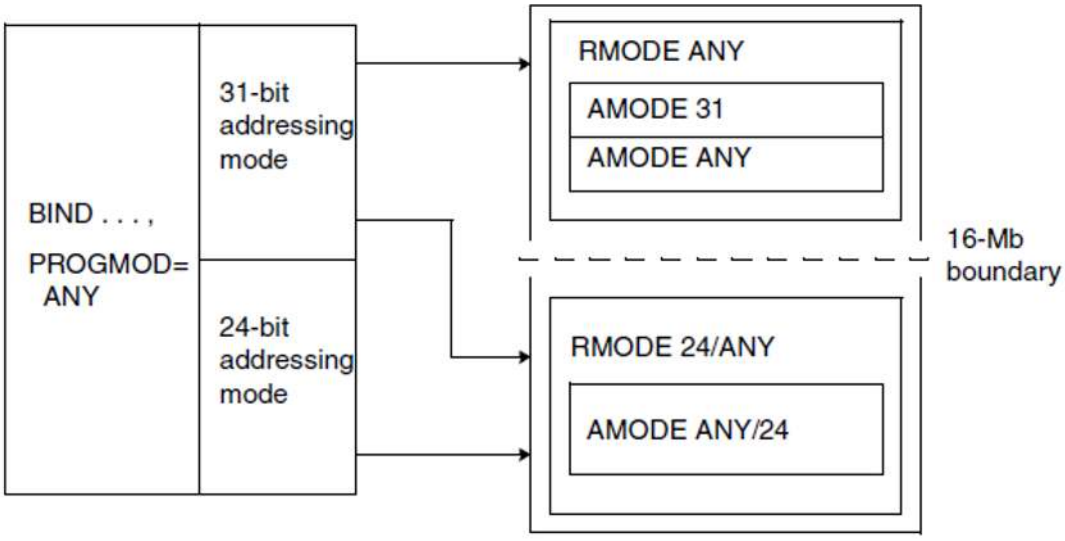


Figure 8: Addressing mode on the BIND macro call

With AMODE-CHECK=*ADVANCED, the following applies:

If after the first module of the load unit has been loaded 24 was determined as AMODE attribute, all other modules of the load unit are loaded below 16 Mb.

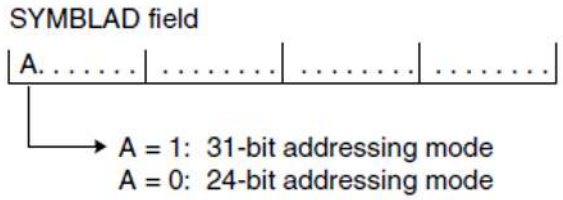
PROGMOD=24

DBL evaluates the operand PROGMOD=24 as follows:

- All modules of the load unit will be loaded below 16 Mb.
- External references will be resolved only if they designate an address below 16 Mb.
- If the operand BRANCH=YES is also specified, 24-bit addressing mode will be set.
- Loading of the load unit will be aborted with an error message if the load unit contains a module with the AMODE 31 attribute.

BRANCH=NO (default)

Once the module has been loaded control is returned to the calling program. The current addressing mode is not changed. Following the BIND macro, the high-order bit of the field that was defined with the SYMBLAD operand indicates which addressing mode has to be set when the module is executed.



This information is determined using the following criteria:

Location of entry point	AMODE (CSECT)		
	24	31	ANY
<= 16 Mb	24-bit mode	31-bit mode	dependent on AMODE set in BIND macro
> 16 Mb	31-bit mode	31-bit mode	31-bit mode

BRANCH=YES

The loaded module is processed first (starting at its entry point). The addressing mode is determined by the value specified for the PROGMOD operand in conjunction with the AMODE attribute of the first control section (CSECT) in the module and taking into account the entry point and the addressing mode set at the time of issuing the BIND macro. The addressing mode is set internally according to the following criteria:

PROGMOD	AMODE (CSECT)		
	24	31	ANY
24	24-bit mode	illegal	24-bit mode
ANY	24-bit mode	31-bit mode	31-bit mode if entry point >= 16 Mb.
			As in BIND macro if entry point < 16 Mb.

3.4.3 Handling COMMON sections

The following points should be noted with regard to the processing of COMMON sections:

- If at least one module of the load unit is loaded below 16 Mb, then all COMMON sections in this load unit will be loaded below 16 Mb.
- The load address of the COMMON section is dependent on the PROG-MOD operand and the RMODE attributes of the COMMON definitions of the same name in the load unit. The load address may possibly also depend on the RMODE attributes of the control section (CSECT) of the same name.

DBL determines the load address of the COMMON in accordance with the PROG-MOD operand as follows:

PROG-MOD	Load address of COMMON
24	< 16 Mb
31	>=16 Mb if all the attributes are RMODE ANY < 16 Mb if at least one attribute is RMODE 24

3.5 Resolving external references above 16 Mb

In BLSSERV up to and including V2.4A the following rules apply for resolving external references:

1. **PROGRAM-MODE=24** in the load command or
   PROGMOD=24 in the BIND macro:

   External references are exclusively resolved with symbols which are loaded below 16 Mb.
2. **PROGRAM-MODE=*ANY** in the load command or
   PROGMOD=ANY in the BIND macro:

   External references are resolved with symbols which are loaded below or above 16 Mb.

If a load unit is loaded with PROGRAM-MODE=*ANY (or PROGMOD=ANY), external references can also be resolved in accordance with these rules with symbols which are loaded above 16 Mb. However, if this load unit has to run in AMODE 24 (for example because of the caller’s AMODE attribut), this results in errors.

To prevent such inconsistencies, additional AMODE checks can be performed when loading with BLSSERV V2.5A or higher. The AMODE-CHECK operand in the START-/LOAD-EXECUTABLE-PROGRAM command (or AMODCHK in the BIND macro) is used to control whether these additional checks are made.

When AMODE-CHECK=*STD (default value), the behavior when loading and resolving external references is compatible with the earlier versions of BLSSERV. BLSSERV decides whether or not external references with modules above 16 Mb can be resolved on the basis of the PROGRAM-MODE operand, the pseudo-RMODE of the loaded objects and the AMODE attribute of the calling program.
This can result in errors occurring like those described above.

When AMODE-CHECK=*ADVANCED, the AMODE attribute of the load unit is also taken into account to decide whether external references can also be resolved with modulesabove 16 Mb. Thus if a load unit receives an AMODE attribute of 24 after its first module has been loaded, the rest of the load unit must be loaded below 16 Mb and external references of this load unit are only satisfied with symbols which can be loaded below 16 Mb.

4 The loader ELDE

- ELDE functions
- Calling ELDE

4.1 ELDE functions

The static loader ELDE also is part of the BLSSERV subsystem. Its function is to load a program (load module) which has been linked by the linkage editor TSOSLNK and stored in a program file or in a program library as a (type C) library element. ELDE also has to evaluate relocation linkage dictionary (RLD) information if the load module does not yet have a core image format (see COREIM=N in the PROGRAM statement of the TSOSLNK linkage editor).

4.2 Calling ELDE

ELDE can be invoked by the user with the following commands and macros:

- The START-EXECUTABLE-PROGRAM or LOAD-EXECUTABLE-PROGRAM command if the FROM-FILE operand has been specified with a file name or if the module to be loaded is a load module.
- The START-PROGRAM or LOAD-PROGRAM command if the FROM-FILE operand has been specified with a file name or with *PHASE.
  The START-PROGRAM and LOAD-PROGRAM will only be supported for compatibility reasons. For new applications the START-EXECUTABLE-PROGRAM or LOAD-EXECUTABLE-PROGRAM should be used.
- The LPOV macro, which dynamically loads overlay segments. The CALL and SEGLD macros implicitly cause the execution of the LPOV macro.

The detailed description of the commands can be found in the „Commands“ manual [5].

5 Migration

This chapter highlights the differences between the old linkage editor/loader system (up to BS2000 V9.5) and the actual Binder-Loader-Starter (BLS) system (as of BS2000 V10.0) and is intended to help the user make the transition.

5.1 Existing concepts and new concepts

First, it is useful to compare and contrast concepts from the old linkage editor/loader system and the actual Binder-Loader-Starter system (see also “Glossary”).

5.1.1 Existing concepts

object module (OM)

Loadable unit which is generated by compiling a source program by means of a compiler.

prelinked object module

Loadable unit which is produced by the TSOSLNK linkage editor by linking together individual object modules (OMs). It is identical in format to an object module.

object module library (OML)

PAM file which contains object modules in the form of library elements. As of LMS V2.0A (SDF format), object module libraries are no longer supported. They 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 a compiler or by the TSOSLNK linkage editor, respectively.

load module (phase)

Executable unit which is produced from object modules (OMs) by the TSOSLNK linkage editor and stored in a cataloged program file or in a program library as a type C library element.

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.

TSOSLNK

Linkage editor of the previous linkage editor/loader system. TSOSLNK links:

- either one or more object modules (OMs) to produce an executable program (load module) or
- multiple object modules (OMs) to produce a single prelinked object module.

DLL

Dynamic linking loader of the previous linkage editor/loader system. It links object modules (OMs) into an executable program and loads this into computer memory.

ELDE

Static loader of the previous linkage editor/loader system. It loads a program (load module) that has been linked by the TSOSLNK linkage editor.

5.1.2 New concepts

link and load module (LLM)

Loadable unit that possesses a logical and a physical structure. It is generated by the linkage editor 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 new linker/loader system.

BINDER

Linker (linkage editor) of the new Binder-Loader-Starter system. It links modules into a link and load module (LLM).

DBL

Dynamic binder loader of the new Binder-Loader-Starter system. It links modules into a load unit and loads this into computer memory.

5.2 Features of the new Binder-Loader-Starter system

The Binder-Loader-Starter (BLS) system, which is available as of BS2000 Version 10, is a significant departure from the concept underlying the old linkage editor/loader system. The key features of the new concept are summarized below.

5.2.1 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 command SHOW-FILE for this.
  In addition to this list output, further easy-to-use information functions are available to the user during a BINDER run.

5.2.2 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 partialapplication 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 up processing when DBL is called repeatedly with the same library.

5.2.3 DBL and BINDER

- One of the main features of the actual Binder-Loader-Starter (BLS) system is the large degree of harmonization between BINDER and DBL. This means:
  - Pseudo-calls, such as the TABLE macro previously used as a link between the static loader ELDE and the dynamic linking loader DLL, are superseded as a result of using LLMs. 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. DBL loads the PRIVATE slice into the user’s class 6 memory. The user declares the PUBLIC slice as shareable by loading it into a common memory pool with the aid of the DBL macro ASHARE.

5.3 Migration from the dynamic linking loader DLL to DBL

- Operating modes RUN-MODE=*STD and RUN-MODE=*ADVANCED
- Incompatibilities in RUN-MODE=*ADVANCED
- Loading LLMs in RUN-MODE=*STD
- Loading an LLM with user-defined slices
- Migration from the old to the new macros

5.3.1 Operating modes RUN-MODE=*STD and RUN-MODE=*ADVANCED

The dynamic binder loader (DBL) runs in two different operating modes. The user selects the mode in which DBL is to execute by means of the RUN-MODE operand in the START-PROGRAM and LOAD-PROGRAM commands.

When the commands START/LOAD-EXECUTABLE-PROGRAM are used, DBL always runs in RUN-MODE=*ADVANCED.

*RUN-MODE=*STD operating mode*
In this mode DBL is fully compatible with the DLL of the old linkage editor/loader system. The new functions in DBL cannot be used in this operating mode. RUN-MODE=*STD is the default setting.

*RUN-MODE=*ADVANCED operating mode*
In this mode DBL supports the functions introduced as of BS2000 V10. These functions can lead to problems of incompatibility with the DLL of the previous linkage editor/loader system.

5.3.2 Incompatibilities in RUN-MODE=*ADVANCED

Searching for modules in libraries

When program libraries are searched for modules, the RUN-MODE operand controls the selection of the modules as follows:

- When RUN-MODE=*STD is set, the search is limited to object modules (OMs). LLMs are ignored.
- When RUN-MODE=*ADVANCED is set, LLMs and object modules (OMs) are compared. In this comparison the element name of an LLM (element type L) or a symbol name in an LLM takes precedence over the element name of an object module (OM) (element type R) or a symbol name in an OM.
- In the commands START/LOAD-EXECUTABLE-PROGRAM it is possible to control the search order by means of the TYPE operand. The order L,C,R is set by default.

This search strategy favors the migration from object module (OM) to link and load module (LLM). Compatibility exists when no LLM is found with the same element name as an OM.

Searching in alternate libraries

By default, the Tasklib is ignored if RUN-MODE=*ADVANCED is set. The optional inclusion of the Tasklib is only possible in connection with the commands START/LOAD-EXECUTABLE-PROGRAM.

Handling name conflicts

How name conflicts are handled is dependent on the selected operating mode (RUN-MODE=*STD or RUN-MODE=*ADVANCED) as follows (see also "Handling name conflicts"):

- When RUN-MODE=*STD is set, name conflicts between CSECTs are always detected, regardless of whether the symbols in a module are masked or not. Name conflicts between masked or nonmasked ENTRYs are always accepted. Only the first ENTRY is used for resolving external references. The user has no means of controlling how name conflicts are handled.
- When RUN-MODE=*ADVANCED is set, name conflicts are detected only if the symbols in a module are *not* masked. The user is informed by means of a message each time there is a name conflict which could cause errors to occur. The NAME-COLLISION operand in the load call gives the user a means of controlling how name conflicts are handled.

5.3.3 Loading LLMs in RUN-MODE=*STD

Many applications, which are no longer being developed, use the LINK macro to dynamically load external runtime components. The migration of such runtime components from the object module to the LLM would therefore have been impossible without sacrificing compatibility, since the LINK macro runs only in the standard RUN-MODE, in which LLMs could not be processed earlier.

To enable migration without losing compatibility, the standard RUN-MODE of DBL was therefore extended so that LLMs can also be processed under certain conditions. In principle, such LLMs must have the attributes of an object module, i.e.:

- it must not include any user-defined slices
- the names of all included OMs must not exceed a length of 8 characters
- no symbol name may be longer than 8 characters
- there must not be any unresolved external reference to a symbol with a name exceeding 8 characters in length
- there must not be any reference from the private LLM section to a PUBLIC symbol with a name that is longer than 8 characters
- the LLM must include information on the logical structure

LLMs which have been processed with the BINDER statement MERGE-MODULES can also be loaded with the LINK macro. Merging enables symbols that are not required to be made invisible externally. The merged LLM or sub-LLM now contains only a prelinked module with one single CSECT.

Due to the differences between the standard and advanced RUN-MODEs, the number of modules that are loaded dynamically with LINK or BIND may differ considerably. Consequently, the loading behavior of both the BIND macro and the LINK macro should be observed very carefully when migrating the runtime components.

5.3.4 Loading an LLM with user-defined slices

The following points should be noted when loading an LLM constructed from user-defined slices:

- the library and the element remain open
- if the LINK-NAME operand is not specified in the load call for the library, the library remains locked for all further load calls.

5.3.5 Migration from the old to the new macros

This section describes the macros which have replaced the macros which were available up to BS2000 V10.0.

LINK macro

The functions of the LINK macro are implemented by the equivalent DBL macro BIND. They are a subset of the functions of the BIND macro.
Note that PROGMOD=ANY is the default value for the BIND macro.
To facilitate migration, the LINK macro has been extended so that LLMs can also be loaded with it under certain conditions (see "Loading LLMs in RUN-MODE=*STD").

LPOV macro

The LPOV macro is not supported for LLMs. When an LLM with user-defined slices is generated by BINDER, no LPOV macro may be contained in any of the included object modules (OMs). Object modules that contain the LPOV macro can only be linked to a program (load module) by the TSOSLNK linkage editor.

PBUNLD and UNLOD macros

The functions of the PBUNLD and UNLOD macros are implemented by the equivalent DBL macro UNBIND. The UNBIND macro is fully compatible with the PBUNLD and UNLOD macros when the operand UNLINK=NO is specified in the UNBIND macro (UNLINK=NO is the default value). In this case CSECTs and ENTRYS are *not* unlinked in the unloaded object, i.e. resolved external references to these symbols remain resolved. When the operand UNLINK=YES is specified in the UNBIND macro, resolved external references are unlinked, i.e. they are treated as unresolved external references. This function is not built into the PBUNLD and UNLOD macros.

VSVI macro

The functions of the VSVI macro are implemented by the equivalent DBL macro VSVI1. The information output is dependent on the selected operating mode (RUNMOD operand).

In the RUNMOD=STD operating mode, the information output by the VSVI1 macro is fully compatible with that produced by the VSVI macro. In this mode the VSVI1 macro processes symbol names and context names up to a maximum length of 8 characters and 16 characters, respectively. Names longer than this are truncated to 8 or 16 characters.The reference size for an item of output information is an entry in the DBL tables with a *fixed* length of 36 bytes.

In the RUNMOD=ADV operating mode, the VSVI1 macro processes names up to 32 characters in length.
The reference size for an item of output information is an entry in the DBL tables with a *variable* length. The length is dependent on the length of the names and the type of information desired.

ITABL macro

The functions of the ITABL macro are a subset of the functions of the VSVI macro and can therefore be replaced by the functions of the VSVI1 macro.

TABLE macro

The functions of the TABLE macro are implemented in extended form in the ETABLE and ETABIT macros. Note that the handling of name conflicts for ETABLE is no longer the same as for the TABLE macro.

5.4 Migrating from the static loader ELDE to the DBL

When calling the loader ELDE you should only use the new commands START/LOAD-EXECUTABLE-PROGRAM. If the load module is present as a C element in a library then these commands should be called without any explicit type specification:

```
START-EXE FROM-FILE=*LIB-ELEM(LIB=library,ELEM=element)
```

or

```
LOAD-EXE FROM-FILE=*LIB-ELEM(LIB=library,ELEM=element)
```

The default setting TYPE=(L,C,R) facilitates a possible subsequent migration to the link load module (LLM). If no explicit DBL parameter declaration is required then it is not necessary to modify the call command on migration. It is enough to enter a corresponding L element in the library.

5.5 Migration of programs to PAM-LLMs

PAM-LLMs differ from (conventional) LLMs because of the “container” in which they are stored:

- a PAM-LLM is stored in a PAM file
- an LLM is stored in a program library as a type L element.

In their internal layout LLMs and PAM-LLMs are identical. Consequently the information in this chapter on migrating programs to LLMs also applies for migrating to PAM-LLMs.

The benefit of migrating programs to PAM-LLMs instead of LLMs is that the commands for the call do not need to be changed: The command `/EXEC HUGO`, `/START-PROGRAM HUGO` or `/START-EXECUTABLE-PROGRAM HUGO` starts the object located in the HUGO file, regardless of whether a program or a PAM-LLM is involved.

6 High-performance loading of programs and products

This chapter provides a description of the measures which should be considered by users in order to optimize the load time for a program/product.

It is generally assumed that the programs or modules are stored in PLAM libraries. This means that the programs /modules are linked with BINDER and loaded with the START-EXECUTABLE-PROGRAM command. The information provided here relates only to working with LLMs.

6.1 General notes on improving loading speed

- All the PLAM libraries involved should be organized as efficiently as possible, i.e.
  - they should contain no redundant LLMs
  - they should be completely reorganized, so that the LLMs they contain are not split into an unnecessary number of extents
  - the library itself should comprise as few extents as possible
  - the library should possibly be created using (STD,2).

  All these measures reduce the number of I/Os and improve runtime behavior. The effectiveness of these measures is, of course, dependent on the original structure. A library created with (STD,2) will be larger than the original library by about 1 KB per member. The reduction in I/O operations will be greater, the more members the library has, provided that these members are not too small. Runtime improvements can then amount to over 10%.
- If modules have the READ-ONLY attribute, one slice should be generated for read-only modules and one slice for read/write modules. This speeds up the loading process by reducing both CPU requirements and I/O operations.
- The number of library accesses is reduced if all symbols not required for subsequent BINDER runs are set to “invisible” with the BINDER statement

  ```
  //MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=...,SYMBOL-TYPE=...,
                             SCOPE=...,VISIBLE=*NO,...
  ```
- If PUBLIC/PRIVATE slices are to be used, format 2 should be used in order to reduce the CPU requirements of BLS and DSSM (see the BINDER statement //SAVE-LLM ... FROM-BS2000-VERSIONS=*FROM-V11).

  The operand SUBSYSTEM-ENTRIES in the BINDER statement //START-LLM-CREATION allows you to specify a symbolic name for the links between the private and public slice. This name must also be defined in the DSSM catalog. (DSSM is called for every link with the old format.)
- You should ensure that the LOGGING parameter in the DSSM parameter file is *not* set to ON (default: OFF). ON would cause DSSM data to be logged when subsystems are loaded.

6.2 Structural measures for reducing resource requirements

Unfortunately, it is rarely possible to quantify precisely the effect of each of the measures described below. The effects depend to a large degree on the structure of the load objects, the subsystems and the libraries. One can, however, generally assume that a gain of at least 10% can be achieved with each of the measures.

Reduce the number of libraries or optimize their contents

- Merge alternative libraries

  If alternative libraries are merged to form a single library (this could even be the specified load library), this results in a considerable reduction in search operations in BLS/PLAM, which in turn leads to a saving in I/O operations and CPU time which is sometimes considerable greater than 10%.

  Merging is generally only recommended if no entries or CSECTs exist with the same name.

- Link in modules from libraries permanently

  If the contents of the alternative libraries and the specified library never or only rarely change, it may be advisable to link the modules into the load object permanently. This reduces the CPU requirements on loading. On the other hand, static linking means that the load object will increase in size, thus generally causing more I/O operations, which in turn increases the runtime.This measure makes sense, therefore, if only small modules are involved and if the alternative libraries contain a particularly large number of entries.

  If you adopt this strategy for improving performance, you must, of course ensure that the link procedure is repeated each time the alternative library is modified.

- Mixed strategy

  If, for instance, the contents of only one of the alternative libraries is modified regularly, it may be advisable to adopt a mixed strategy using both of the methods described above.

Restricting dynamic loading of shared code

- Do not dynamically load/link shared code

  Many programs do not need to dynamically load/link from a subsystem/shared programs or user shared memory. These programs should be started as follows:

  ```
  /START-EXECUTABLE-PROGRAM ...,RESOLUTION=(SHARE-SCOPE=*NONE),...
  ```

  This prevents BLS from searching the subsystems for the appropriate entry, resulting in an often considerable reduction in CPU time.

- Only dynamically load from system memory

  This is the default. It is not possible to restrict the search for the entry performed by DSSM to specific subsystems or to shared programs only.

- Only dynamically load from user shared memory

  When you issue the START-EXECUTABLE-PROGRAM command, you can issue the parameter

  ```
  RESOLUTION(...,SHARE-SCOPE=*MEMORY-POOL(...)...)
  ```

  to restrict loading to user shared memory. It is also possible to further restrict the operation to specified memory pools.

- Preload user shared memory

  As far as possible, all shareable sections of the applications in use should be preloaded (e.g. FOR1-RTS). This means that they are loaded only once and only need to be linked (with little overhead) for each further load operation. This strategy generally results in a saving of considerably more than 10% in I/O operations and CPU requirements.

Eliminating ESD information

- Complete elimination

  This measure only applies for standalone programs. These are LLMs which are not called by other programs and which are completely prelinked. For programs of this type, the load operation can be accelerated if the ESD and other BLS information tables are eliminated using the appropriate BINDER call. Call BINDER as follows:

```
/START-BINDER
//START-LLM-UPDATE LIB=<orig-lib>,ELEMENT=<elem-name>
//SAVE-LLM LIB=<new-load-lib>,TEST-SUPPORT=*NO,MAP=*NO,
           SYMBOL-DICTIONARY=*NO,LOGICAL-STRUCTURE=*NO,
           RELOCATION-DATA=*NO
//END
```

  This parameter specification means that only those BLS structures are created which are required for loading a simple program, thus reducing the size of the object. This saves CPU time and I/O operations when the program is loaded. All the information required for other processing is then no longer available. This means that

  - these parameters cannot be used for load objects which are split into a number of slices
  - no LLM updates can be carried out on an object of this type.
  - relocatable objects (e.g. as part of a runtime system) cannot be created.

  The (supplementary) parameter setting REQUIRED-COMPRESSION=*YES should not be selected, since, although up to 10% of I/O operations can be saved, approx 40% more CPU time is required.

- Partial elimination

  When merging LLMs or sub-LLMs to form a prelinked module with a single CSECT, you should specify what ESD information is to remain in the external address book with the BINDER statement MERGE-MODULES. This can result in a considerable reduction in BLS overhead during loading.

6.3 Improving the load speed for C programs

You can improve the load speed of C programs by linking the few modules which cannot be preloaded from the C runtime system to the compiled program. Use the following supplementary BINDER statement:

```
//RESOLVE-BY-AUTOLINK LIB=<Partial-Bind-Lib>
```

The <Partial-Bind-Lib> is installed under the name SYSLNK.CRTE.PARTIAL-BIND. It is also necessary to make external names invisible in order to suppress duplicate symbols:

```
//MODIFY-SYMBOL-VISIBILITY SYMBOL-NAME=*ALL,VISIBLE=*NO
```

These two statements result in a considerable reduction in CPU time and particularly in runtime when loading C programs. The load object is only a few pages (approx. 18) longer than the compiled program.

Note

The CRTEC and CRTECOM subsystems must be loaded to allow CRTE to be loaded in the shared code area.

6.4 Use of DAB

If the linked object and/or the libraries used are very large, with the result that a large number of I/O operations are required during loading, you should check whether it is possible or necessary to use DAB (Disk Access Buffer).

DAB permits such a considerable reduction in I/O operations on frequently accessed files that in extreme cases the runtime depends only on the CPU requirements.

A cache area must be selected for the load object which is the same size as the load object itself. This is particularly simple if the object is the only member of the library. You can then issue the command

```
/START-DAB-CACHING AREA=*FILE(FILE-AREA=<library>),CACHE-SIZE=*BY-FILE
```

to reserve a main memory area with the same size as the library. This is filled the first time the object is loaded, with the result that no time-intensive I/O operations are required for any subsequent load operations.

The cache areas for the libraries can generally be created smaller than the total size of the libraries, since it is generally the case that not all of the modules contained in the libraries will be dynamically loaded. In this case, the cache size must be specified explicitly (in KB or MB) when the /START-DAB-CACHING command is issued. It is possible to monitor the cache hit rate for each cache area using SM2 or the /SHOW-DAB-CACHING command. The hit rate should be at least 80%. If this is not the case, the size of the cache area must be increased.

7 Glossary

This glossary contains definitions of key terms used in the description of the Binder-Loader-Starter (BLS) system. Cross-references are indicated by *italic* typeface 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)
- 32-bit addressing (AMODE=32)
- 24-, 31- or 32 bit addressing (AMODE=ANY).

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 memory pool

A memory area in class 6 memory (user memory) which may be shared and accessed by several users.

COMMON section

Data area which can be shared by a number of *control sections (CSECTs)* for communication purposes.

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*. This 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 or *prelinked modules* produced by the linkage editor TSOSLNK 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 point (ENTRY)

Symbolic link address which is defined in one *module* but can also be used by another module.

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)

external symbols vector (ESV)

external symbols vector (ESV)

Contains all the *program definitions* and *references* in a *module* and is required for resolving references.
ESV records are contained in *link and load modules (LLMs)*, whereas *object modules (OMs)* contain ESD (external symbol dictionary) records.

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 *ILE 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.

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 it is stored 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.

list name unit

If several or all objects of a library are loaded with a single call of the BIND macro, a list name unit is created. This prevents the need for multiple DBL calls when loading a symbol list of the same load unit.

load module

Synonym for *program*

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.
LRLD records are contained in *link and load modules (LLMs)*, whereas *object modules* contain RLD (*relocation linkage dictionary*) records.

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 information

Information in a *link and load module (LLM)* which defines the *logical structure* of the LLM.

logical structure of an LLM

Defines the tree structure of a *link and load module (LLM)*. The root is formed by the *internal name*, the nodes are formed by *sub-LLMs*, and the leaves are formed by *object modules (OMs)* and empty *sub-LLMs*.

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 compiler.

object module library (OML)

PAM file which contains *object modules* as library elements.

PAM-LLM

LLM which has been saved 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*.

prelinked object module

Loadable unit which is linked by the TSOSLNK linkage editor from individual *object modules (OM)*; it has the same format as an object module (OM).

program

Executable entity which is linked by the TSOSLNK linkage editor from *object modules (OMs)* and stored in a cataloged program file or as a type C library element (cf. *element type*) in a *program library*.

program definition

Generic term for

- *control section (CSECT)*
- *entry point (ENTRY)*
- *indirect linkage entry (ILE)*
- *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)*

relocation linkage dictionary (RLD)

Information in a *module* which determines how addresses are to be adjusted relative to a common base address during linking and loading.
RLD records are contained in *object modules*, whereas *link and load modules (LLMs)* contain LRLD (*local relocation dictionary*) records.

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).

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).

server module

ILE server.

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.

8 Related publications

The manuals are available as online manuals, see http://manuals.ts.fujitsu.com, or in printed form which must be paid and ordered separately at http://manualshop.ts.fujitsu.com.

[1] **BINDER**
Binder in BS2000 OS DX
User Guide

[2] **ASSEMBH**
Reference Manual

[3] **LMS**
SDF Format
User Guide

[4] **BS2000 OS DX**
System Installation
User Guide

[5] **BS2000 OS DX**
Commands
User Guide

[6] **JV**
Job Variables
User Guide

[7] **BS2000 OS DX**
Executive Macros
User Guide

[8] **AID**
Advanced Interactive Debugger
Core Manual
User Guide

[9] **BS2000 OS DX**
Introduction to System Administration
User Guide

[10] **DSSM/SSCM**
Subsystem Management in BS2000 OS DX
User Guide

[11] **BS2000 OS DX**
Utility Routines
User Guide

[12] **BS2000 OS DX**
Introductory Guide to DMS
User Guide

[13] **SDF**
Introductory Guide to the SDF Dialog Interface
User Guide

[14] **SDF-P**
Programming in the Command Language
User Guide

[15] **SDF-A**
User Guide

[16] **LMS**
ISP-Format
User Guide

[17] **XHCS**
8-Bit Code Processing in BS2000 OS DX
User Guide

[18] **POSIX**
POSIX Basics for Users and System Administrators
User Guide