

**The z/OS Program Management Binder
and Its “Program Object” Format:
What The New Program Model
Will Mean to You**

SHARE 102 (Feb. 2004), Session 8170

John R. Ehrman
ehrman@us.ibm.com or ehrman@vnet.ibm.com

IBM Silicon Valley (nee Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, CA 95141

© IBM Corporation 1995, 2004. All rights reserved.

Table of Contents

Contents-1

Topic Overview	1
The Binder and Program Loader: Overview	2
The z/OS Binder and Program Loader	3
Program Management Data and Control Flow	4
Binder and Program Loader: History	5
Binder Features	7
Program Loader Features	9
A Brief Review of Old Object and Load Modules	10
Traditional Terminology	11
Translator Output: Object Modules	12
Object Module External Symbol Dictionary (ESD)	13
Old External Symbol Types and Ownership Hierarchy	14
Assembler Example of Object Module External Symbols	15
Load Modules: A “Refresher” View	16
New Executable Structures: Program Objects	17
New Terminology for Program Objects	18
A Program Object: Some Basic Definitions	19
Sections	20
Classes	21
Class Attributes	22
Sketch of a Multi-Class Program Object	23
Benefits of Demand-Loaded (NOLOAD) Classes	24

Table of Contents**Contents-2**

Compatibility	25
New External Symbol Types and Ownership Hierarchy	26
Program-Object Mapping of Old Object/Load Modules	27
Example of Object Mapping at Assembly Time	28
Program-Object Mapping of Object/Load Modules	29
Mixed-Mode Modules and RMODE(SPLIT)	30
RMODE(SPLIT) Program Object	31
Improved Binding Techniques	32
Binding Attributes and Rules	33
Example of External Data MERGE Binding	35
Generalized Address Constants	37
Binder Inputs and Outputs	38
Module Data: Binder Input (Logical View)	39
Module Data: Binder Output (Logical View)	40
Module Data: PMLoader Input (Physical View)	41
The Generalized Object File Format	42
What Is a "GOFF"?	43
Generalized Object File Format Records	44
High Level Assembler GOFF-Support Options and Statements	45
High Level Assembler Support for Class Attributes	46
High Level Assembler CATTR Usage	48
Example: A Simple Two-Class Assembler Language Program	49
ESD From Simple Two-Class Assembly	50

z/OS Binder and Program Objects

© IBM Corporation 1995, 2004. All rights reserved.

Table of Contents**Contents-3**

Dynamic Link Libraries (DLLs)	51
Dynamic Linking and Dynamic Link Libraries	52
Dynamic Linking: Preparation and Use	53
Dynamic Linking: Execution Time	54
Summary, Glossary, and References	55
Comparing Old and New	56
Summary	57
Glossary and Definitions	58
References	64

z/OS Binder and Program Objects

© IBM Corporation 1995, 2004. All rights reserved.

1. The z/OS MVS Program Management Binder and Program Loader
 - New features in z/OS R3
2. Brief review of old object and load modules
3. New executable-module structures: Program Objects
 - All about Sections, Classes, Elements, and Parts
 - How Program Objects are like and unlike Load Modules
4. Compatibility with old Object and Load Modules
5. New treatments of familiar binding techniques
6. The Generalized Object File Format
7. Dynamic Link Library support
8. Glossary and References

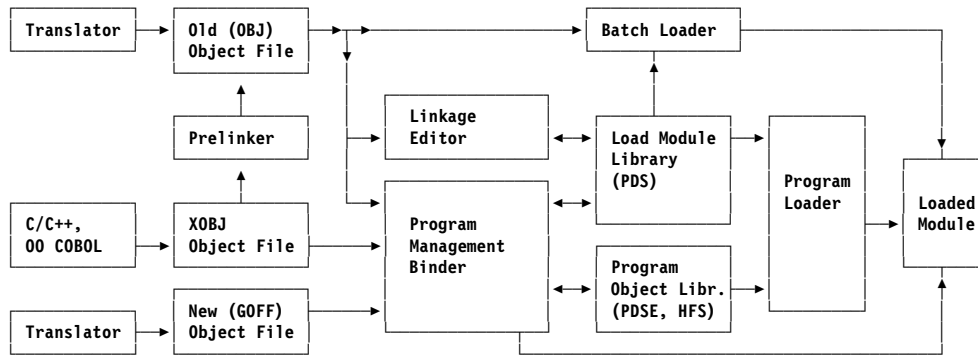
Note: This is NOT a tutorial on Binder usage!

**The Binder and Program Loader:
Overview**

Some useful abbreviations...

PM	Program Management
LM	Load Module
PO	Program Object
OM,OBJ	Object Module (Traditional Format)
GOFF	Generalized Object File Format

- New technology
 - **Binder** replaces Linkage Editor, Batch Loader;
 - **Program Loader** (PMLoader) extends and generalizes Program Fetch
- Answers a large set of customer requirements
 - Many new options, messages, added information, detailed diagnostics
 - Almost all internal constraints and “Table Overflow” conditions eliminated
- Creates **Program Objects** (a new form of “executable”)
 - Supports long names, multiple text classes, new adcon types, new object file formats
 - “Linear” format permits efficient “DIV” mapping directly to virtual storage
 - Stored in PDSE's (which fix almost all PDS problems: space, integrity, compression, performance, shareability, etc.), or in HFS
 - “Split-RMode” modules allow separation of code/data text blocks by RMode
- Base for all future enhancements
 - Linkage Editor and Batch Loader are “Functionally Stabilized”
 - Subsumes C/C++ Prelinker functions
 - PO format not externalized (APIs give access to all data)



Note: Arrowheads indicate direction of data flow.
 ↔ means a component can be produced as output or read as input.

- LMs reside only in PDSs; POs reside only in PDSEs or HFS files
- Can mix OM and GOFF to produce PO or LM (LM restricts features)
 - “Source → OM → LKED → LM” equivalent to “Source → OM → Binder → LM”
 - “Source → OM → LKED → LM” equivalent to “Source → GOFF → Binder → LM”
- Can bind PO and LM to produce either (LM restricts features)

PM1 DFSMS/MVS V1R1: "Modern" program management

- New Binder and Program Loader
- Support for PDSE libraries
- Linkage Editor compatibility support

PM1.1 DFSMS/MVS V1R2: Support for HFS**PM2** DFSMS/MVS V1R3:

- Enhanced PO structure
- Split-RMODE modules, distributed loading
- GOFF/ADATA support
- Fast-path data retrieval API

PM3 DFSMS/MVS V1R4:

- Binder includes C/C++ Prelinker functions, new options, control statements
- Support for DLLs (including HFS, Archive files)
- Dynamic Linklib and Dynamic LPA support for PDSEs

PM3.1 OS/390 V2 R10: XPLINK support

- Mangled/demangled names table, external-symbol and HFS-file attributes

PM4 z/OS V1R3: BCP component 5695PMB01, FMID HPM7706

- 64-bit virtual support: AMODE(64), 8-byte adcons, quadword alignment
- External-name maximum length extended to 32K
- Saves dynamic-bind information across rebinds
 - INCLUDE -ATTR, -IMPORTS, -ALIASES copies info from input module
- Reduced PO size (mainly for C/C++ programs)
- Default PO format: minimum needed to support options/features in use
- New manuals: old Binder manual split into
 - z/OS MVS Program Management: User's Guide and Reference (SA22-7643)
 - Introductory material, options, control statements, JCL
 - Over 200 changes, 37 new graphics, 45% pages changed
 - z/OS MVS Program Management Advanced Facilities (SA22-7644)
 - Programming interfaces, data areas, record formats
 - Over 250 changes, 26 new graphics, 30% pages changed
- Load modules support AMODE(64), 8-byte adcons (no quad alignment)

PM4.2 z/OS V1R5:

- Improved error recovery via new ESPIE exit, especially for APIs
- More data retained about origins of program object's components
- Initial RMODE(64) support for C_WSA64

- External names to 32K bytes
 - Character set X'41'-X'FE', plus SI/SO; optional case sensitivity
 - Long names OK for autocall, control statements, APIs, all resolutions
- POs support multiple text classes, total text length up to 1GB
 - Uniform treatment of Associated Data (“ADATA”), other non-loaded classes
- Supports new **Generalized Object File Format**, OBJ, and XOBJ
 - **GOFF**: produced by C/C++ and High Level Assembler; defined by Binder
 - **OBJ**: traditional Object Module
 - **XOBJ**: produced by C/C++, OO-COBOL; extension of OM
 - Binder converts XOBJ internally to GOFF format; output of bind must be a PO
- Extended support for OS/390 Unix System Services

- Prelinker elimination enhances usability, efficiency
 - Rebindable output: no need to relink from object
 - Simpler service: can ship only the necessary object files
- Integrated processing for specialized C/C++ features
 - C370LIB, HFS archive files for autocall resolution
 - Prelinker control statements, renaming, new classes, mangled names, etc.
 - LE runtime routines load (non-reentrant) Writable Static Area (WSA)
- Dynamic Link Libraries (DLLs) (more at slides 51-54)
 - New functions in Binder, Program Loader, LE, Contents Supervision
 - Defer linking/loading to run-time decisions
- Binder Interface Exit
 - Allows modifying existing resolutions, renaming, forcing new autocall search

- Page-fault loading (“page mode”) or pre-loaded (“move mode”)
 - Page mode (default):
 - POs *mapped* into virtual storage using Data In Virtual (DIV), except from HFS files
 - Entire module virtualized if shorter than 96K bytes, or if bind option FETCHOPT=PRIME was specified
 - Otherwise, segments (up to 64K each) virtualized as referenced
 - Faster initiation, less central storage allocated “immediately”
 - Move mode:
 - Preloads and maps entire module in intermediate storage, then moves to destination
 - Accommodates directed loads, “packed” modules, overlay, V=R
- Can load/delete “deferred-load” classes on request
- POs (including DLLs and deferred-load classes) can be staged in LLA
- PDSEs, POs, DLLs support and exploit “Dynamic LPA”
- Under Unix System Services, POs in HFS are written/read as “flat files”

A Brief Review of Old Object and Load Modules

Compatibility with new formats described at slides 25- 31
(Old formats detailed in Session 8169)

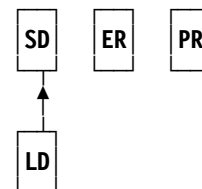
- Control Section (**CSECT**) (Was often called just a “Section”)
 - The basic indivisible unit of linking and text manipulation
 - A collection of program elements bearing **fixed** positional relationships to one another; its addressing and/or placement relative to other Control Sections does not affect the program's run-time logic
 - Ordinary (**CSECT**) and Read-Only (**RSECT**) have machine language text; Common (**COM**) and Dummy (**DSECT**) have no text
- External Symbol (“public”; internal symbols are “private”)
 - A name known at program linking time, whose value is intentionally not resolved at translation time
- PseudoRegister (or, External Dummy Section)
 - A special type of external symbol whose value is resolved at link time to an **offset** in an area (the “PRVector”) to be instantiated during execution
- Address Constant (“Adcon”)
 - A field within a Control Section into which a value (typically, an address) will be placed during program binding, relocation, and/or loading

- Five types of (card-image) records:
 - ESD** External Symbol Dictionary (C/C++ generates a variant, **XSD**)
 - TXT** Machine Language instructions and data (“Text”)
 - RLD** Relocation Dictionary (for address constants)
 - SYM** Internal Symbols
 - END** End of Object Module, with **IDR** (Identification Record) data
- At least one control section per object module
- “Batched” translations may produce multiple object modules
- For the fascinating details, see:
 - High Level Assembler for MVS & VM & VSE *Programmer's Guide*, SC26-4941
 - z/OS MVS Program Management Advanced Facilities*, SA22-7644

- Describes four basic types of **external symbols**:
 - SD,CM** **Section Definition**: the name of a control section (Blank-named control section called “**Private Code**,” **PC**; zero-length PC sections discarded by LKED/binder)
 - LD** **Label Definition**: the name of a position at a fixed offset within a Control Section; typically, an Entry Point. (The only symbol type having no ESDID of its own)
 - ER,WX** **External Reference**: the name of a symbol defined “elsewhere” to which this module wants to refer (**WX** = “**Weak EXternal**”; not a problem if it's unresolved)
 - PR** **PseudoRegister**: the name of a PseudoRegister (The Assembler calls it an “**EXternal Dummy Section**,” **XD**)
PR names are in a separate “name space” from all other external symbols, and may duplicate non-PR names without conflict.
- Two external symbol scopes: library (SD, LD, ER); module (PR, WX)

- Four external symbol types:
 - SD** Section Definition: owns LDs
 - LD** Label Definition: entry point within an SD; no ESDID of its own
 - ER** External Reference
 - PR/XD** PseudoRegister/External Dummy: this section's view of (contribution to) the PRV
- Lack of ownership of ER and PR items can cause problems when relinking
- Contrast this with the new (slide 26)

Old External Name Ownership Hierarchy



- A program with each symbol type:

```

Sect_A Start 0 (SD)
      DC 5D'0.1'
      Entry A_Entry (LD)
A_Entry DC Q(My_XD)

      Extrn External (ER)
      Wxtrn Weak_Ext (WX)
MyCom COM , (CM)
      DS 12D

My_XD DXD 3D (XD)

Sect_B Csect (SD)
      Entry B_Data (LD)
B_Data DC 7D'1.0'

      Csect , (PC)
      DC A(MyCom)
      End Sect_A
    
```

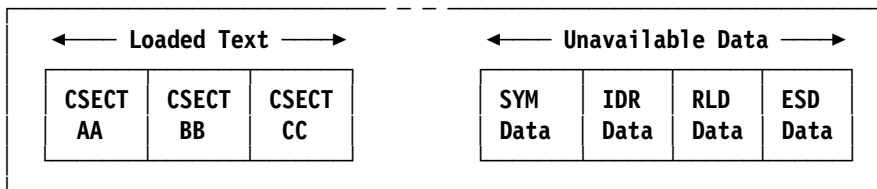
- External Symbol Dictionary:

Symbol	Type	Id	Address	Length	LD ID
SECT_A	SD	00000001	00000000	0000002C	
A_ENTRY	LD		00000028		00000001
EXTERNAL	ER	00000002			
WEAK_EXT	WX	00000003			
MYCOM	CM	00000004	00000000	00000060	
MY_XD	XD	00000005	00000007	00000018	
SECT_B	SD	00000006	00000000	00000038	
B_DATA	LD		00000000		00000006
PC	PC	00000007	00000000	00000004	

- A_ENTRY is in SECT_A (LD ID = 1), at offset X'28'
- Private Code has blank section name

- Contrast with new format (slide 28)

- Load modules have a one-dimensional “block format” structure:



- All loaded text has a single set of attributes
 - One RMODE, one AMODE; entire module is R/W or R/O (“RENT”)
 - All text is loaded relative to a single relocation base address
 - Effectively, a single-component module
- Other module data not accessible via “normal” services

New Executable Structures: Program Objects

New Terminology for Program Objects

- Some new terms are introduced, some old terms are used differently
 - No “Control Sections” in a PO (traditional CSECTs are mapped to **elements**)
- **Section**: a “handle” or a “cross-section”
 - Neither a CSECT name nor an external name
 - Used in control statements to manage Binder actions
- **Class**: attributes are important; name is rarely referenced
- **Element**: indivisible unit of text (analogous to an OM/LM CSECT)
- **Part**: Multiple identically-named external-data definitions are “merged”
 - Translator-defined **Part References** (PRs) are bound into Program Object **Part Definitions** (PDs)
- Five ESD symbol types: SD, ED, PR, LD, ER (see slide 26)
 - Compared to OM's four:
 - SD different; ED new; PR generalized; LD, ER same (see slide 14)
 - **PR** can now mean either “Part Reference” or “PseudoRegister”
 - Four symbol scopes: Section (new), Module, Library, Import-Export (new)
- Two **binding attributes** and binding methods: **CAT** and **MRG**

- Most easily visualized as a **two**-dimensional structure:

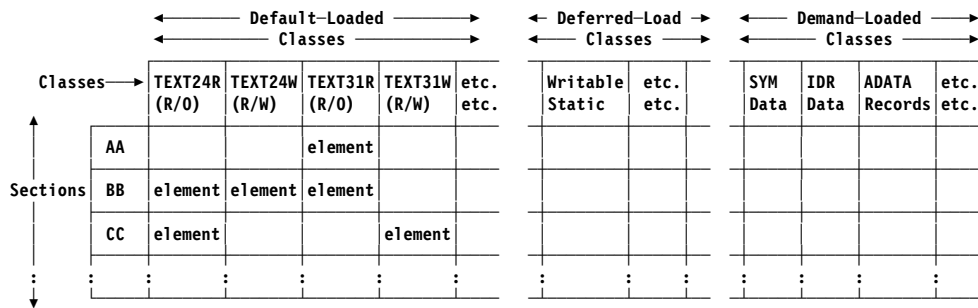
	Class X	Class Y	Class Z
Section A	Element	Element	Element
Section B	Element	Element	Element

- One dimension is determined by a **section name**
 - Analogous to OM Control Section name (but not the same!)
- Second dimension is determined by a **class name**
 - Analogous to a loadable module's name (but not the same!)
 - Attributes (e.g. RMODE) assigned to each class (more at slide 22)
- The unit defined by a section name and a class name is an **element**

- A **section** is the program unit manipulated (replaced, deleted, ordered, or aligned) by user control statements during binding
 - Operations on a section apply to all elements within the section
 - Including rejection! (Only the first occurrence of a section is kept)
- Each **section** may supply contributions to one or more **classes**
 - According to their desired binding and loading characteristics
 - Assembler Language example (slide 49) illustrates this
- Section names must be unique within a Program Object
 - As for Load Modules
 - **Note:** Section names are not external names or implied labels
 - Not used to resolve external references
 - Label Definitions (LDs) within elements are used to identify positions in text
- Binder-created sections “own” module-level data
 - E.g. ESD data, class maps, SYM data, module-level ADATA, Part Definitions
 - Your code should avoid section names starting with **IEWB** (see slide 40)

- Each class has uniform loading/binding characteristics and behavior
 - All section contributions to each class are bound together in a **segment**
 - More than one class may have identical attributes (e.g., RMODE(31))
 - Binder may put classes with identical attributes into one segment (Thus, class offsets may be different from segment offsets)
- Class loading characteristics determine the load-time placement of the segments in virtual storage
 - Loadable segments are loaded as separately relocated non-contiguous entities
 - Not all segments are normally loadable (e.g. IDR)
 - POs may have multiple class segments (each analogous to a Load Module!)
- Class names (max. 16 characters) are purely mnemonic, and are rarely externalized
 - Naming conventions provide for class sharing, and avoid class-name collisions among independent compilation units
 - Names of the form letter_symbol are **reserved!**
 - Example: names like C_xxx reserved to compilers, B_xxx to Binder
 - B_MAP describes names and contents of each class
 - B_ESD contains external names
 - B_IMPEXP contains imported/exported external names (for DLL support)

- Separate attributes may be assigned to each class, such as:
 - **RMODE**: indicates placement in virtual storage of a loaded segment
 - Loadability
 - **LOAD**: The class is brought into memory when the program is initially loaded
 - Same as Load Module's usual behavior
 - **NOLOAD**: The class is not loaded with the program; may not contain adcons
 - Non-text classes are always NOLOAD; application loads via Binder API
 - **DEFERRED LOAD**: The class is prepared for loading, instantiated when requested
 - Useful for byte-stream data such as pre-initialized private writable static data areas in shared (re-entrant) programs
 - Text type: **Byte-stream** (machine language) or **Record-like** (IDR, ADATA)
- **AMODE** assignable to entry points
- Other attributes are accepted by the Binder for future use:
 - Read-only/Read-write; Movable/Nonmovable; Shareable/Nonshareable; REFR/REUS/RENT



- All elements in each class have identical behavioral attributes (e.g., RMODE)
- Each loaded class segment has its own relocation origin
 - Effectively, a *multi-component* (multi-LM?) module! (compare slide 16)
- Demand-loaded (NOLOAD) classes accessible via Binder services
- Deferred-load classes require special Program Loader interface

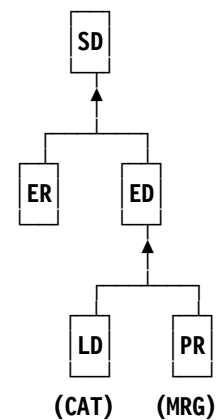
- Integrated, optional support for any type of program-related data
 - IDR data, translator's "Associated Data" (ADATA), user data
- PO can keep module-related and user data together in one place
 - *Optionally*, of course! As much or as little as desired
 - Source statements (possibly encoded), source-file information, etc.
 - Internal symbols, debugging breakpoint tables, NLS messages, etc.
 - User information, history data, documentation, instructions, etc.
- Application requests data via Binder's "FASTDATA" API
 - Delivers what was "Unavailable Data" in Load Modules
- Allows problem determination and debugging "in place"
 - Helps tools locate bugs when and where they happen
- Reduces need for complex configuration management tools
 - Module-specific items (source, object, listings, executables) need not be tracked separately

Compatibility

New External Symbol Types and Ownership Hierarchy

- Five external symbol types:
 - SD** Section Definition; owns other types
 - ED** Element Definition: defines the class to which this element (and its text, parts, and/or labels) belongs; owned by an SD
 - LD** Label Definition: entry point within an element; owned by an ED; only in CAT classes; has own ESDID and AMODE (unlike OBJ)
 - PR** Part Reference or PseudoRegister: this section's view of a contribution to an item within a class; owned by an ED; only in a MRG class
 - ER** External Reference: owned by an SD
- Strict ownership rules prevent orphaned symbols (OBJ has orphans; see slide 14)

New External Name Ownership Hierarchy



- All functionality of old OM/LM behavior is retained
- Old code is mapped by the Binder as follows:

OM	Binder's Mapping
SD	SD; create ED for class B_TEXT and LD at element's origin for section name
LD	LD
ER, WX	ER, WX
CM	SD with "common" flag; create ED for class B_TEXT and LD at element's origin for section name
PC	Binder assigns unique numeric names (displayed as \$PRIVnnnnnn)
PR, XD	PR; create ED for class B_PRV (special PseudoRegister class)
TXT	Text records
RLD	RLD records
END	END; deferred length (if any) placed on a new record type
SYM	ED for class B_SYM

- Assembler supports similar mappings when GOFF option is specified...
- IEBCOPY of LM (PDS) to PO (PDSE) invokes the Binder

- Sample program:
(based on slide 15 example)

```

Sect_A  Start 0          (SD)
        DC 5D'0.1'
        DC Q(My_XD)

MyCom   COM ,          (CM)
        DS 12D

My_XD   DXD 3D         (XD)

Sect_B  CSect ,        (SD)
        Entry B_Data    (LD)
B_Data  DC 7D'1.0'

        End Sect_A
    
```

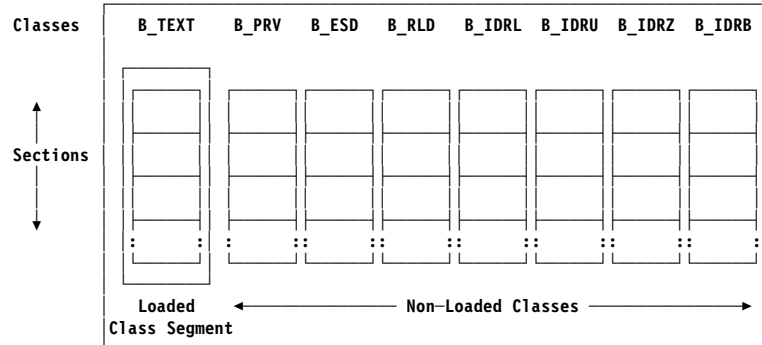
- OM ESD (HLASM **OBJECT** option)

Symbol	Type	Id	Address	Length	LD ID
SECT_A	SD	00000001	00000000	0000002C	
MYCOM	CM	00000002	00000000	00000060	
MY_XD	XD	00000003	00000007	00000018	
SECT_B	SD	00000004	00000030	00000038	
B_DATA	LD		00000030		00000004

- GOFF ESD (HLASM **GOFF** option)

Symbol	Type	Id	Address	Length	LD ID
SECT_A	SD	00000001			
B_PRV	ED	00000002			00000001 (new)
B_TEXT	ED	00000003	00000000	0000002C	00000001 (new)
SECT_A	LD	00000004	00000000		00000003 (new)
MYCOM	SD	00000005			
B_PRV	ED	00000006			00000005 (new)
B_TEXT	ED	00000007	00000000	00000060	00000005 (new)
MYCOM	CM	00000008	00000000		00000007 (new)
MY_XD	XD	00000009	00000007	00000018	
SECT_B	SD	0000000A			
B_PRV	ED	0000000B			0000000A (new)
B_TEXT	ED	0000000C	00000030	00000038	0000000A (new)
SECT_B	LD	0000000D	00000030		0000000C (new)
B_DATA	LD	0000000E	00000030		0000000C

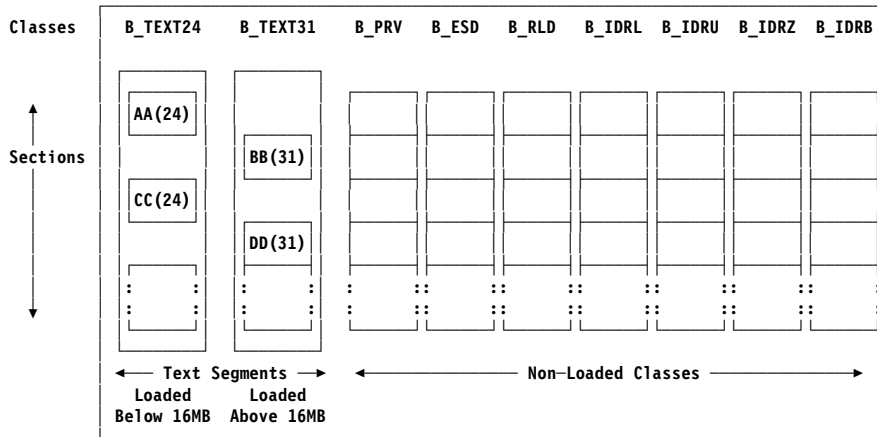
- Old modules are mapped into POs (if SYSLMOD is a PDSE):



- B_TEXT “Loaded Class” behaves like traditional LM's text
- B_ESD is like LM CESD; B_RLD is like LM Control/RLD records
 - B_IDRx classes hold IDR data from Language translators (L), User (U), Super Zap (Z), and Binder (B)

- Link Editor: linking modules with mixed RMODEs forces the LM to most restrictive value
 - Only way to split a program into RMODE(24) and RMODE(31) parts:
 - Link them separately; execute one part, which loads the other
 - No external-symbol references are resolved between the two modules! (LOAD/LINK only know entry point name and address of loaded module)
- Binder: **RMODE(SPLIT)** option creates a PO with two text classes
 - Affects only class B_TEXT:
 - RMODE(24) CSECTs (from class B_TEXT) moved to TEXT_24 class, RMODE(31) CSECTs (from class B_TEXT) moved to TEXT_31 class
 - TEXT_24 class loaded below 16M, TEXT_31 class loaded above 16M
 - Supports full capabilities of inter-module external symbol references
 - As if entire program was linked as a single LM in “most restrictive” style!
 - Internal-symbol inter-class references usable (see example at slide 49)
 - Simple solution to LM's AMODE/RMODE complexities
 - User code must handle addressing-mode switching, if any is needed
- Recommendation: let the Binder determine AMODEs and RMODEs

- Binder “splits” B_TEXT class into two RMode(24)/RMode(31) classes



- Inter-class references resolved automatically
- Easiest if program runs uniformly in AMode(31)

Improved Binding Techniques

- Link Editor binding algorithms
 - Retained
 - Generalized
 - Treated more rigorously

- Classes have one of two **binding attributes**: Catenate, Merge
 - Determines algorithms used to map and bind the **class segments**

1. **Catenate** (CAT)

- Section contributions (**elements**) are aligned and catenated end-to-end
 - The familiar manner of text binding
 - Zero-length elements are retained (but take no space)
- Ordering determined in the normal manner
- **Note:** Only the first element with a given section and class name is retained; subsequent identically-named sections are rejected (same as LKED's CSECT rejection)

2. **Merge** (MRG)

- A generalization of LKED/LDR binding of CM, PR items
- Section contributions to MRG classes are Commons, Pseudo-Registers, and Parts
 - Each section supplies its own view of any number of shareable external data items
- Commons and Pseudo-Registers are “overlaid” in Merge binding (they map the same storage)
- Binding parts in MRG classes:
 - a. Determine longest length, most restrictive alignment
 - b. Text inputs from later parts “overlay” text from earlier parts
- Parts are accessible to any section referencing the part
- **Note:** All Part information is retained for accurate re-binding

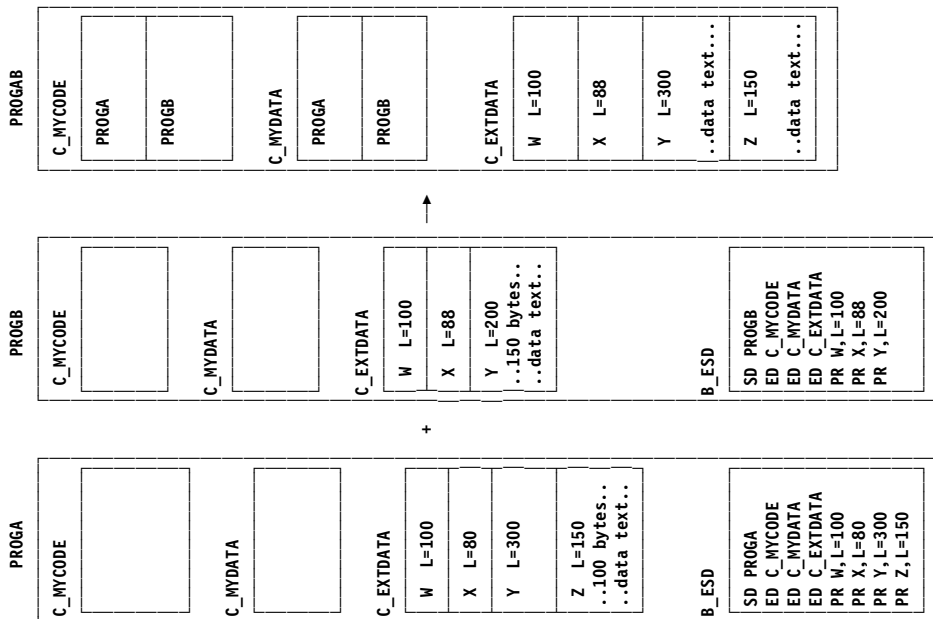
- Programs PROGA and PROGB are bound to form PROGAB:
 - In addition to the C_MYCODE and C_MYDATA **CAT** classes, the two programs have each defined external data (Part Reference) items in **MRG** class C_EXTDATA:
 - PROGA has defined four Part-Reference items: W, X, Y and Z.

SYMBOL	DEFINED LENGTH	
W	100	
X	80	
Y	300	
Z	150	(contains initializing text)

- PROGB has defined three Part-Reference items: W, X, and Y.

SYMBOL	DEFINED LENGTH	
W	100	
X	88	
Y	200	(contains initializing text)

- If initial text was provided for W, X, Y, or Z, it would be saved in class B_PARTINIT to enable correct re-binding
- Only one instance of initializing text is retained; all Parts with text must have identical length (but not alignment)
- In the next figure, only compiler-defined text/ESD classes are shown
 - The resultant ESD for PROGAB is a combination of the two input ESD items (and has been omitted to improve readability)

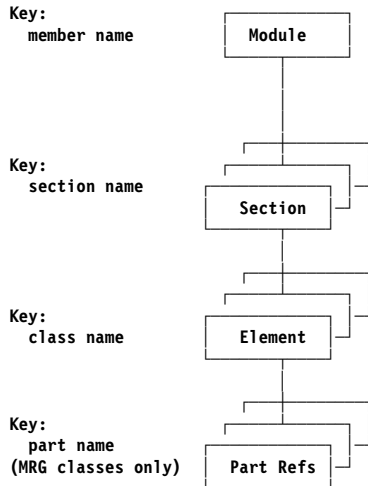


- Address of “Writable Static” (non-shared, private work area)
 - Implemented in Assembler Language as R-type address constant
- Length of any class or part
 - Generalization of “Cumulative External Dummy” (CXD, length of PRV)
 - Implemented in Assembler Language as J-type address constant
- Offset of a part or label within its class
 - Generalization of Assembler's Q-type address constant
- Binder/Loader “Token”
 - Used for requesting PMLoader virtualization of DEFERRED LOAD classes
 - Not externalized



Binder Inputs and Outputs

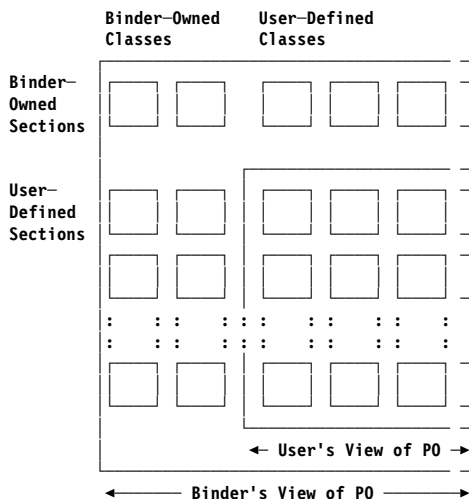
- Some pictorial views of binding and loading



PO structure as seen by the translator and Binder user:

- **Section** roughly equivalent to a “compilation unit”
 - Consists of **elements** in various classes
- MRG classes are constructed from **Part References** and **PseudoRegisters**

Binder Output view is more complex!



Text classes are bound into **segments**

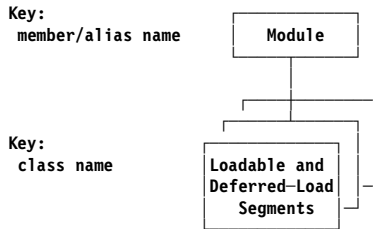
- A segment may contain multiple classes if they have identical attributes

Binder retains extra “module-level” data for re-bindability

- PR items (and any initializing text) (class B_PARTINIT)
- control information (e.g. B_ESD)
- IDR data, module map, etc.

in reserved section names like

- X'00000001' for B_ classes, orphaned ER or PseudoRegister items
- X'00000003' for PDs, linkage descriptors, initializing data
- IEWBLIT for LE support (class B_LIT)
- IEWBCE for DLL support (class B_IMPEXP)



PO structure seen by PMLoader:

- PO consists of one or more **class segments**, some of which are loadable by default or on request
- PMLoader loads and relocates segments
 - Each segment is like a LM: relocated with its **own** origin address
 - **Distributed** or **scatter** loading
- Library member names (entry points and aliases) must be in same “primary” class segment as the module entry point

**The Generalized Object File
Format**

- Documented in *z/OS MVS Program Management Advanced Facilities*, SA22-7644

- **Generalized Object File Format:** replacement for old Object Module
 - Generated by High Level Assembler for most architected functions
 - C/C++ implementation starting with OS/390 V2R10
- **Supports needs of languages, PO structure, Binder**
 - long external names
 - 32-bit length and offset fields (vs. 24 in OM)
 - multiple text classes
 - up to 1 million (or more) ESDIDs and external symbols
 - user and associated data (ADATA) in object stream
 - ...and many other forms of attributes and descriptive data

- **Six record types (similar to the five OM types)**
 1. **Module Header (new):** CCSID, translator product identification, etc.
 2. **External Symbol Dictionary:** long names, rich set of types and attributes
 3. **Text:** object code, IDR, ADATA
 - OM: IDR only on END; ADATA only in text or a side file
 4. **Relocation Dictionary:** relocation information
 5. **Deferred Element Length (new)**
 - In case anyone still uses this old OM END-record function
 6. **End:** with optional Entry-Point nomination
- **Open-ended, flexible architecture; allows growth and expansion**

- GOFF option creates a GOFF file
 - Existing, unmodified code will go into special “compatibility” classes
 - B_TEXT for text, B_PRV for pseudoregisters (see slides 27-28)
 - Requires LIST(133) option for wide listing format
- Section names specified with START, CSECT, RSECT
- CATTR statement defines class name, specifies Class ATTRibutes:


```
classname CATTR attribute[,attribute]...
```

classname

a valid PO class name; it must follow the rules for naming external symbols, except that:

- class names are restricted to a maximum of 16 characters
- all class names of the form *letter_symbol* are reserved for IBM-defined purposes

attribute

binder attributes to be assigned to the class

- XATTR statement declares additional external-symbol attributes

- Attributes currently supported by the Binder:

ALIGN(<i>n</i>)	Aligns class elements on a 2^n boundary ($0 \leq n \leq 12$) Currently: for text, 3, 11, or 12; for PVs, 0-3
MERGE	The class has the merge binding attribute (default = CAT)
NOLOAD	The class is not loaded when the PO is brought into storage (default = LOAD)
DEFLOAD	Requests deferred loading of the class
RMODE(24)	The class has residence mode 24
RMODE(31)	The class has residence mode 31
RMODE(ANY)	The class may be placed in any addressable storage; equivalent to RMODE(31)

- Attributes currently accepted (but not supported) by the Binder:

MOVABLE The class is reenterable, and can be moved
(It is adcon-free, and can be mapped to different virtual
addresses in different address spaces)

EXECUTABLE, NOTEXECUTABLE (or null)
The class can/cannot be branched to or executed;
null operand means “unspecified”

READONLY The class should be storage-protected

REFR The class is marked refreshable

RENT The class is marked reenterable

REUS, NOTREUS
The class is marked reusable or not

- CATTR must be preceded by START, CSECT, or RSECT
 - A section name must be defined first
 - Unlike OM, no blank section is initiated
 - Following text belongs to the element defined by the section and class names
- If several CATTR instructions have the same class name:
 - the first occurrence establishes the class and its attributes
 - the rest indicate the continuation of the class, and may not specify attributes
- Default attributes for CATTR (if none are specified) are:

ALIGN(3), NOTREUS, RMODE(24)

- Same as the assembler's OM defaults

- The module defines one section (Sect_A), two classes (Code24, Code31):

```

Sect_A Csect,      Start of section 'Sect_A'
Code24 CAttr RMode(24),Executable Define 'Code24' Class
***** Portion loaded below 16MB
Entry Start      Declare entry point name
AMode 24         Entry point has AMODE(24)
Using *,15       Establish addressability
Save (14,12),*   Save registers
                ...set up save areas etc.
LR 12,15        R12 is base register
Drop 15         Drop old base
Using Start,12  Establish addressability
                Finish init'z'n code
                L 15,=A(X'80000000'+MainCode) Point to Code31
                BASSN 14,15 Call MainCode
                LiOrg
D31Addr DC A(Data31) RMode(24) Literal pool
Data24 DC ...      Addr(data above 16M)
                ...data below 16M...
Code31 CAttr RMode(31),Executable Define 'Code31' Class
***** Portion loaded above 16MB
Using *,15       Establish base regs etc.
MainCode Save (14,12),* 'MainCode' is INTERNAL!
D24Addr DC A(Data24) Addr(data below 16M)
Data31 DC ...     ...data above 16M...
                End Start Nominate 'Start' entry
    
```

- Note inter-element references using internal symbols!
- Note AMODE for entry-point name: LD items have AMODEs, sections don't (classes have RMODEs)
 - LDs in a section needn't all have the same AMODE!

- The assembled example creates this ESD listing:

External Symbol Dictionary

Symbol	Type	Id	Address	Length	LD ID	Flags	(Annotations)
SECT_A	SD	00000001					(Section definition)
B_TEXT	ED	00000002	00000000	00000000	00000001	00	(Default class; length=0)
SECT_A	LD	00000003	00000000		00000002	00	(Label for section)
CODE24	ED	00000004	00000000	00000074	00000001		(User class)
START	LD	00000005	00000000		00000004	01	(Label in CODE24; AMODE(24))
CODE31	ED	00000006	00000078	00000012	00000001		(User class)

- Section SECT_A (SD) “owns” elements (ED) in three classes:
 - B_TEXT “owns” the label (LD) for SECT_A
 - created by HLASM because it doesn't know if other classes will be defined
 - CODE24 “owns” the label (LD) for START
 - CODE31 has no externally visible labels
- LD ID column shows “Owning ID”
- HLASM requires the **GOFF** option for this to work

Dynamic Link Libraries (DLLs)

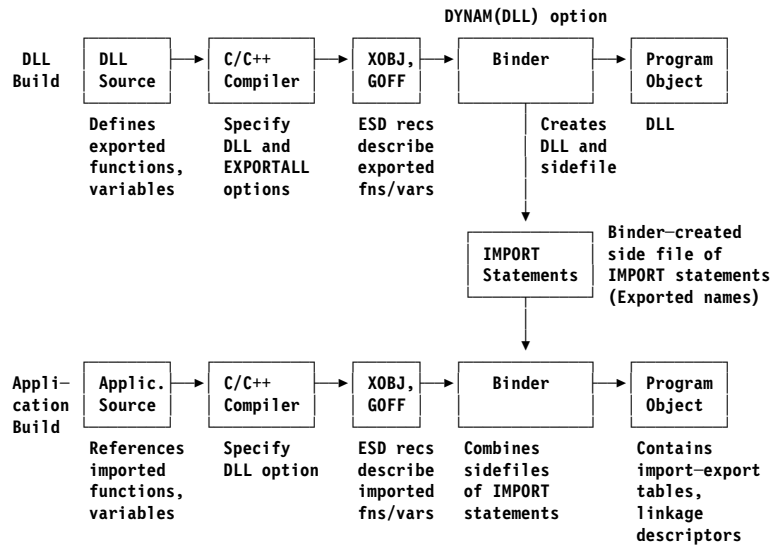
Dynamic Linking and Dynamic Link Libraries

- Dynamic linking: binding of external names at execution time
 - DLLs provide one form of dynamic linking; LE is required
- DLL creator identifies names of functions and variables to be **exported**
 - Makes them available in a “side file” for runtime binding to other applications
 - Compiler indicates “import-export” status in object file
- DLL-using application identifies functions and variables to be **imported**
 - User must specify compiler DLL option and Binder control statements
- Binder also provides the IMPORT control statement

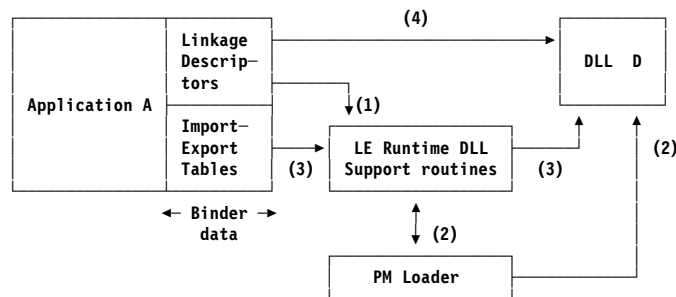
```
IMPORT CODE|DATA,d11_name,identifier
```

 - Compilers and HLASM XATTR statement declare IMPORT/EXPORT status
- Binder creates side file, import-export tables and linkage descriptors
 - DYNAM(DLL) option required for DLL creator and user
- LE runtime support routines load and link specified names

- Example using C/C++: create a DLL, then the application



- Example: Application A imports names from DLL D:



- (1) First reference to an imported name passes control to LE
- (2) LE DLL-support routines invoke PM Loader to load the DLL
- (3) Linkage to DLL name is completed:
 - LE uses import-export table to update descriptors for code/data items
 - Different "linkages" are used for code (functions) and data (variables)
- (4) Subsequent application references go directly to the requested (imported) name in the DLL
 - Linkage Descriptors updated to provide direct reference

**Summary, Glossary, and
References**

Comparing Old and New

- Binder and PMLoader support both Old and New:

	Old (Load Modules)	New (Program Objects)
Components	Link Editor, Program Fetch, Batch Loader	Binder, Program Loader
Library	PDS	PDSE, HFS
Executables	One-dimensional; single AMODE, RMODE	Two-dimensional; multiple segments and A/RMODEs
Size limit	16MB	1GB
Symbols	8 characters	32K characters
Symbol types	SD, LD, ER, PR	Same, plus ED
Module info	IDR only; no system support	Any data; Binder API
DLL support	Prelinker required	Integrated
Extensibility	Not possible	Open-ended architecture

- New technology for MVS “executables”
 - Efficient storage and loading
 - Flexible program segmentation
 - Generalized mechanisms for inter-component references
- Satisfies many requirements from customers, languages, operating systems and hardware
- Retained (but non-obtrusive) information about programs
- Application Programming Interfaces to all functions/data
- Open-ended designs for all items
 - Easy to generalize, enhance and improve
 - Enables Program Management evolution to meet future requirements
- **For You:** Much more flexibility in creating program structures

- ADATA** Associated Data: program data stored in a PO which is not required for binding, loading, or execution.
- API** Application Programming Interface
- CAT** A binding method whereby section elements in a class are aligned and concatenated.
- CCSID** Coded Character Set ID: identifies a character set used in an assembly or compilation.
- class** A cross-section of Program Object data with uniform format, content, function, and behavioral attributes.
- Common** A CSECT having length and alignment attributes (but no text) for which space is reserved in the Program Object (see Part View)
- compilation unit**
A “fresh start” of a translator's symbol tables. There may be more than one compilation unit per source input file.

deferred load

A class attribute requesting the PMLoader prepare the class (a Prototype Section, or "PSect") for rapid loading on request during execution. (Usually, for non-shared classes.)

distributed loading

See "scatter loading"

element The unit of program object data uniquely identified by a class name and a section name.

external data

Module data accessible by multiple sections, each defining its own view as a Part View.

GOFF Generalized Object File Format, a new and extensible object file supporting Binder and PMLoader features.

linear format

The format of a PO, "loaded" by DIV mapping.

loadable A class attribute indicating that the class is to be loaded with the program object.

load module (LM)

The original form of MVS executable, stored in record format.

MRG A binding method whereby identically named PR items in a class are merged and aligned before catenation with other PR items.

no load A class attribute indicating that the class may be "demand loaded" by the application.

Part Binding

In a MRG class, using CAT binding

Part Reference (PR), PseudoRegister (PR), External Dummy (XD)

A named subdivision of a MRG class, a PseudoRegister or external data item (Part), having length and alignment attributes. Resolved at an offset within the class segment. Space in the loaded module is reserved for Parts, but not for Commons or PseudoRegisters.

-
- PM1** The Binder, Loader and related program management services available in DFSMS/MVS V1R1.0 and V1R2.0. Emulates Linkage Editor/Loader function; simple PO structure.
- PM2** Extensions to the program management services delivered with DFSMS/MVS V1R3.0. Significant modifications and enhancements to PM1 PO structure.
- PM3** Extensions to the program management services which became available with DFSMS/MVS V1R4.0. Significant modifications and enhancements to PO structure and function.
- PM3.1** In OS/390 V2R10; XPLINK support, other extensions
- PM4** In z/OS V1R3; 64-bit virtual, 32K names, reduced PO size, saved dynamic link info, enhanced archive-file and C370 library support
- PM4.2** In z/OS V1R5; improved error recovery, retained data about program object components, initial RMODE(64) support
-

program object

The new form of MVS executable, stored in linear format.

record format

The format of a LM, loaded by Program Fetch I/O operations.

relocation

The load-time conversion of address constants from module or class displacements to virtual addresses.

scatter loading

The loading of module text into non-contiguous areas of virtual storage according to class attributes stored with the module. Also referred to as *distributed loading*.

- section** (1) A cross-section of Program Object data stored under a single name. A section consists of elements belonging to one or more classes. (2) A generic term for control section, dummy section, common section, etc.; a collection of items that must be bound or relocated as an indivisible unit.
- segment** The aggregate of all section contributions to a single class, stored in consecutive locations on DASD and (optionally) loaded as a single entity into virtual storage. Each segment has its own relocation base address.
- text** (1) The class(es) of module data containing the machine language instructions and data. (2) A class attribute indicating that locations within the class may contain and/or be the target of address constants.

1. z/OS MVS Program Management: User's Guide and Reference (SA22-7643)
2. z/OS MVS Program Management Advanced Facilities (SA22-7644)
3. DFSMS/MVS V1R3.0 Presentation Guide (GG24-4391), chapter 6
4. "Linkers and Loaders," by Leon Presser and John R. White, *ACM Computing Surveys*, Vol. 4 No. 3, Sept. 1972, pp. 149-167.
5. Linkage Editor and Loader User's Guide, Program Logic manuals

These publications describe the Assembler Language elements that create inputs to the Linkage Editor, Loader, and Binder.

6. High Level Assembler for MVS & VM & VSE Language Reference (SC26-4940)
7. High Level Assembler for MVS & VM & VSE Programmer's Guide (SC26-4941)