The DFSMS/MVS Binder and Its "Program Object" Format: What The New Program Model Will Mean to You

SHARE 96 (Feb. 2001), 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

March 1, 2001

Table of Contents Contents-1

Table of Contents Contents-2

Table of Contents Contents-3

Topic Overview

Note: This is NOT a tutorial on Binder usage!

1. The DFSMS/MVS Binder and Program Loader
   - Replacing the Linkage Editor and Batch Loader
2. 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

The Binder and Program Loader: Overview

Some useful abbreviations...

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

Binder and Program Loader: History and Terminology 3

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: some items require OS/390 V2R4, LE 1.8

- 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

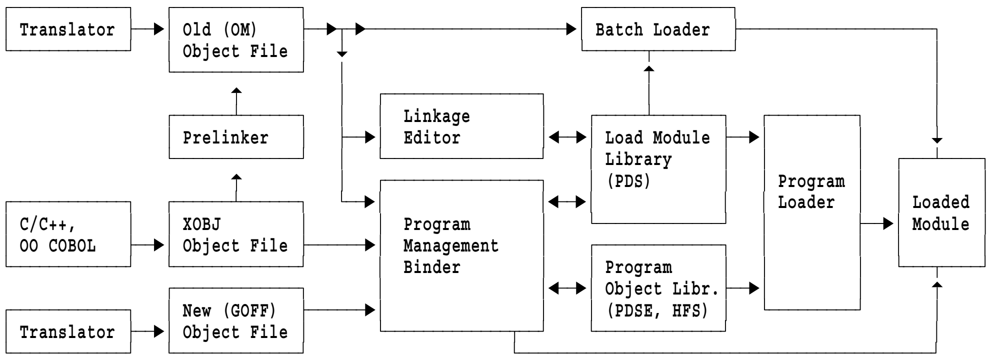
DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

The DFSMS/MVS Binder and Program Loader 4

- Totally new product and new technology
  - **Binder** replaces Linkage Editor, Batch Loader;
    Program Loader (PMLoader) extends and generalizes Program Fetch
  - Requires OS/390 and DFSMS/MVS
- Answers a large set of customer requirements, including a vast array of usability and performance problems
  - 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, and much more!
  - “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
- Base for <u>all</u> future enhancements
  - Linkage Editor and Batch Loader are “Functionally Stabilized”

DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

Program Management Data and Control Flow 5



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 → GOFF → Binder → LM"
- Can bind PO and LM to produce either (LM restricts features)

Binder Features 6

- External and module-alias names to 1024 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
  - "Split-RMode" modules allow separation of code/data text blocks by RMode
  - Uniform treatment of Associated Data ("ADATA"), other non-loaded classes
- Supports new **Generalized Object File Format**, OM, and XOBJ
  - **GOFF**: produced by C/C++ and High Level Assembler; defined by Binder
  - **OM**: 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

Binder Features ... 7

- 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 52-55)
  - 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

DFSMS/MVS Binder and Program Objects

Program Loader Features 8

- POs ***mapped*** into virtual storage (except from HFS files)
- Page-fault loading (“page mode”) or pre-loaded (“move mode”)
  - Page mode (default):
    - Mapped into virtual storage using Data In Virtual (DIV)
    - Entire module mapped if shorter than 96K bytes,
      or if bind option FETCHOPT=PRIME was specified
    - Otherwise, segments (up to 64K each) mapped as referenced
  - 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”

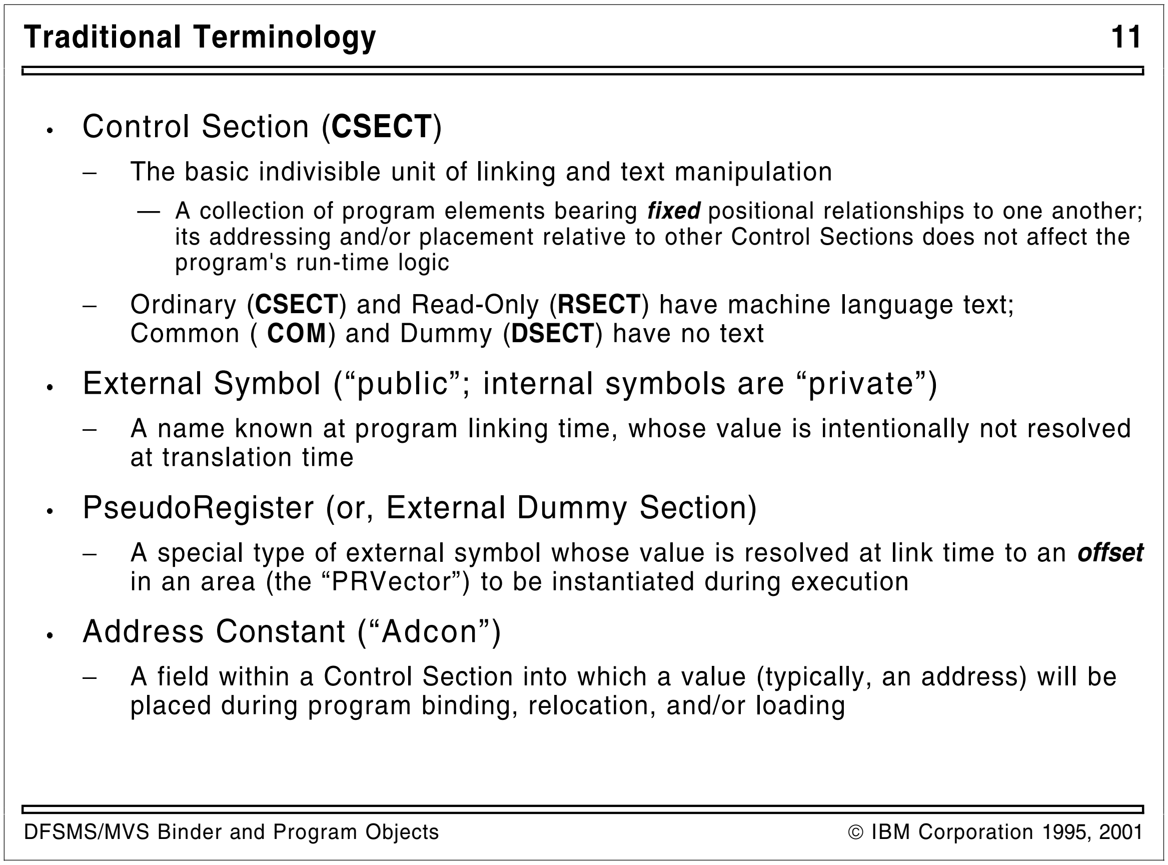
DFSMS/MVS Binder and Program Objects

Program Object Format Is Not Externalized 9

- Binder API eliminates need to understand Program Object format
  - Allows format changes for new hardware and software technology
    - Three different PO formats have already been used!
    - Old PO formats loadable by current and future versions of PMLoader
  - Future format enhancements will be transparent to users
- API supports input and retrieval of all PO data
  - All PO data available (including user data, “ADATA”)
  - “FastData” API for read-only access to PO data
- PM supports “transportable” format for Program Objects
  - Recommend using IEBCOPY to move POs among MVS data sets!
  - IEWTPORT service creates format resembling buffers of text, RLDs, IDRs, CESDs, etc. accessed using the Binder Interface
    - Output usable for reconstructing Program Objects
  - Use is now deprecated

10

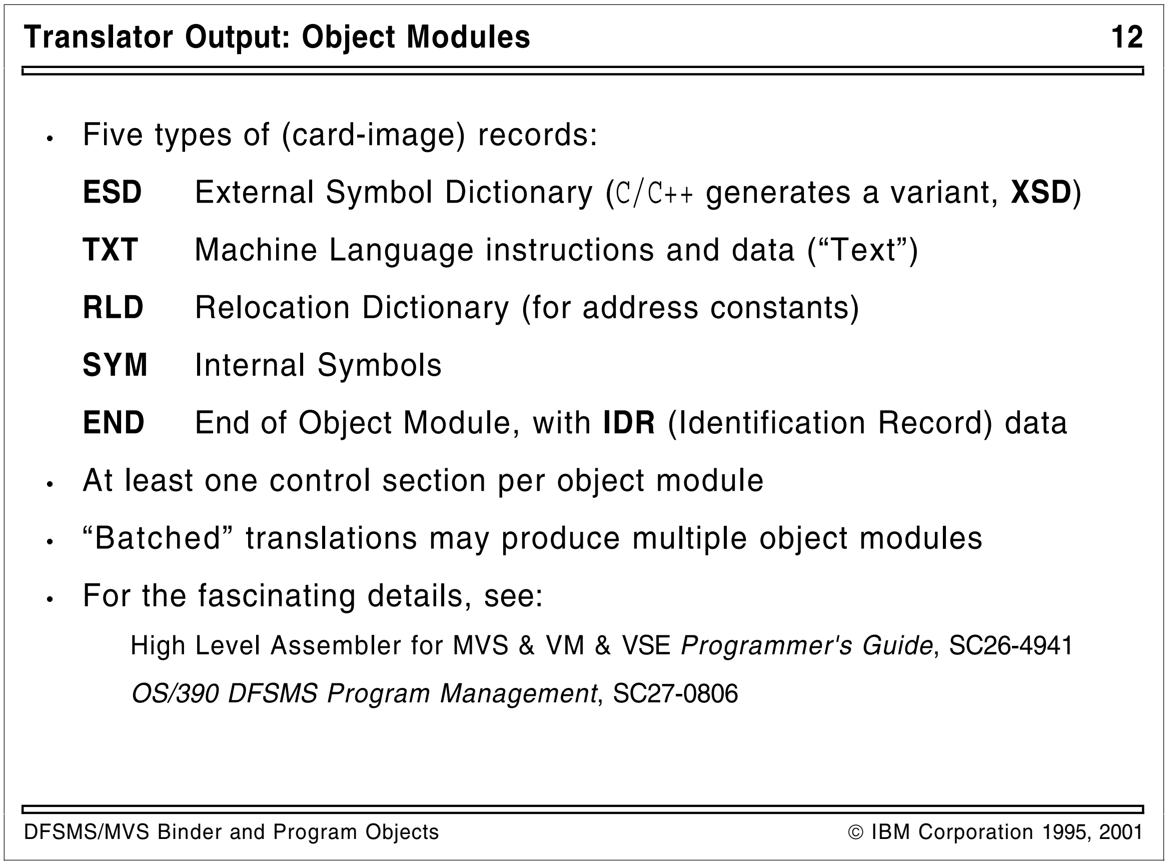
A Brief Review of Old Object and Load Modules



Traditional Terminology 11

- Control Section (**CSECT**)
  - 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

DFSMS/MVS Binder and Program Objects



Translator Output: Object Modules 12

- 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

  OS/390 DFSMS Program Management, SC27-0806

DFSMS/MVS Binder and Program Objects

Object Module External Symbol Dictionary (ESD) 13

- 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**)

  LD Label Definition: the name of a position at a fixed offset within a Control Section; typically, an Entry Point

  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 match non-PR names without conflict.

- Two external symbol scopes: library (SD, LD, ER); module (PR, WX)

Old External Symbol Types and Ownership Hierarchy 14

- Four external symbol types:

  SD Section Definition: owns LDs

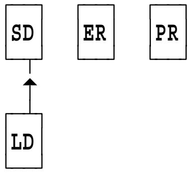
  LD Label Definition: entry point within an SD

  ER External Reference

  PR/XD Pseudo-Register/External Dummy: this section's view of (contribution to) the PRV

- Lack of ownership of ER and PR items can cause problems when relinking
- We will contrast this with the new (at slide 25)

Old External Name Ownership Hierarchy



Assembler Example of Object Module External Symbols 15

- 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)
         DC    7D'1.0'

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

- External Symbol Dictionary:

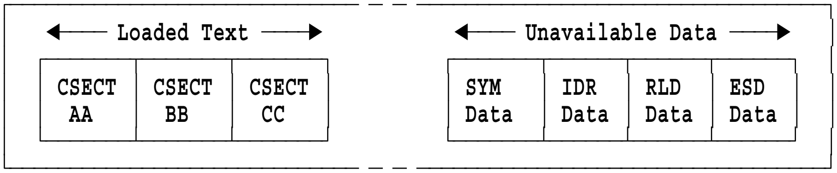
```
Symbol   Type  Id      Address  Length    LD ID

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 00000030 00000038
          PC 00000007 00000068 00000004
```

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

Load Modules: A “Refresher” View 16

- 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 (CSECTs are mapped to **elements**)
- ***Section***: a “handle” (*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 CSECT)
- ***Part***: Commons and PseudoRegisters (and initializing text)
  - Translator-defined **Part Views** (PVs) are bound into Program Object **Part Definitions**
- Five ESD symbol types: SD, ED, PV (or PR), LD, ER (see slide 25)
  - Compared to OM's four: SD different; ED new; PR generalized; LD, ER same (see slide 14)
  - Four external symbol scopes: section, module, library, import-export
- Two ***binding attributes*** and ***binding methods:*** CAT and MRG
  - Linkage Editor used both, but less rigorously (details at slide 32)

A Program Object: Some Basic Definitions 19

- 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 (see slide 22)
- The unit defined by a section name and a class name is an ***element***
  - Viewable as the “intersection” of a section and a class

Sections 20

- 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!
- Each ***section*** may supply contributions to one or more ***classes***
  - According to their desired binding and loading characteristics
  - Assembler Language example (slide 50) 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. class maps, SYM data, module-level ADATA, Part Views
  - User code should avoid section names starting with **IEWB**..!

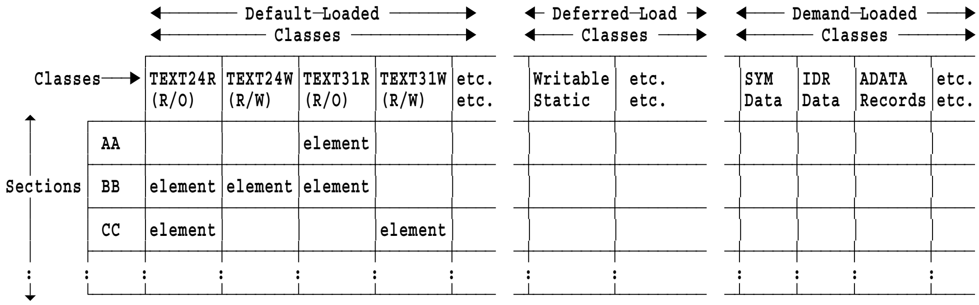
Classes 21

- 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 (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. 14 characters) are purely mnemonic, and are rarely externalized
  - Naming conventions required for class sharing, and to 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)

Class Attributes 22

- 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)
- Other attributes are accepted by the Binder for future use:
  - Read-only/Read-write; Movable/Nonmovable; Shareable/Nonshareable

Sketch of a Multi-Class Program Object 23



		Default-Loaded Classes					Deferred-Load Classes		Demand-Loaded Classes			
	Classes	TEXT24R (R/O)	TEXT24W (R/W)	TEXT31R (R/O)	TEXT31W (R/W)	etc. etc.	Writable Static	etc. etc.	SYM Data	IDR Data	ADATA Records	etc. etc.
Sections	AA			element								
	BB	element	element	element								
	CC	element			element							
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

- New concept: separately relocatable ***classes*** of module data
  - 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

Benefits of Demand-Loaded (NOLOAD) Classes 24

- Integrated support for any type of program-related data
  - IDR data, translator's “Associated Data” (ADATA), user data
- PO keeps module-related and user data together in one place
  - 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

New External Symbol Types and Ownership Hierarchy

25

- Five external symbol types:

  SD Section Definition; owns other types

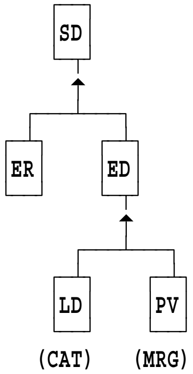
  ED Element Definition: defines the class to which this element (and its text, parts, and/or labels) belongs; owned by an SD

  LD Label Definition: entry point within an element; owned by an ED; only in a CAT class

  PV Part View: this section's view of (contribution to) a part 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



26

Compatibility

Treatment of Old Object/Load Modules 27

- 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
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 PRIVnnnnn)
PR, XD	PV; create ED for class B_PRV
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...

DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

Example of Object Mapping at Assembly Time 28

- Sample program:

```
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
```

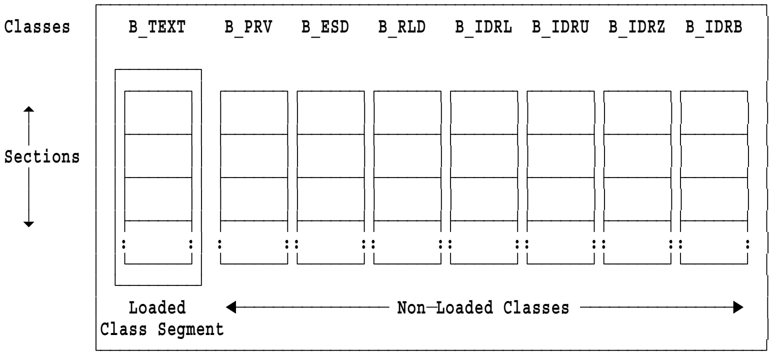
- GOFF ESD (HLASM **GOFF** option)

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

DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

Program-Object Mapping of Object/Load Modules 29

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



- B_TEXT “Loaded Class” behaves like traditional LM's text (see slide 16)
- B_ESD is like LM CESD; B_RLD is like LM Control/RLD records
  - B_IDRx classes hold IDR data from **L**anguage translators (L), **U**ser (U), Super**Z**ap (Z̄), and **B**inder (B)

  IEBCOPY of PDS to PDSE invokes the Binder to do the conversions

Mixed-Mode Modules and RMODE(SPLIT) 30

- 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 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 50)
  - Simple solution to LM's AMODE/RMODE complexities
    - — User code must handle addressing-mode switching, of course!
  - Without RMODE(SPLIT) option, B_TEXT is bound as with LMs
- Recommendation: let the Binder determine RMODEs

Improved Binding Techniques

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

Binding Attributes and Rules

Classes have one of two ***binding attributes***:

1. ***Catenate*** (CAT)
   - Section contributions (***elements***) are aligned and catenated end-to-end
     - The familiar manner of text binding
   - Ordering determined in the normal manner
   - **Note:** Only the first element with a given section and class name is retained; others are rejected (same as LKED's CSECT rejection)
2. ***Merge*** (MRG)
   - A generalization of LKED/LDR binding of CM, PR items
   - Section contributions are named ***Part Views*** (PVs)
     - Each section supplies its own *view* of any number of shareable external data items
   - Views are “overlaid” in Merge binding (they map the same storage)
     - Part Views are bound into ***Part Definitions*** (PDs) (see examples on slides 34-35)
     - Part Definitions subsequently bound into segments by catenation
   - Parts are accessible to any section referencing the part
   - **Note:** All Part Views are retained, whether or not identically named

Example of External Data MERGE Binding 33

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

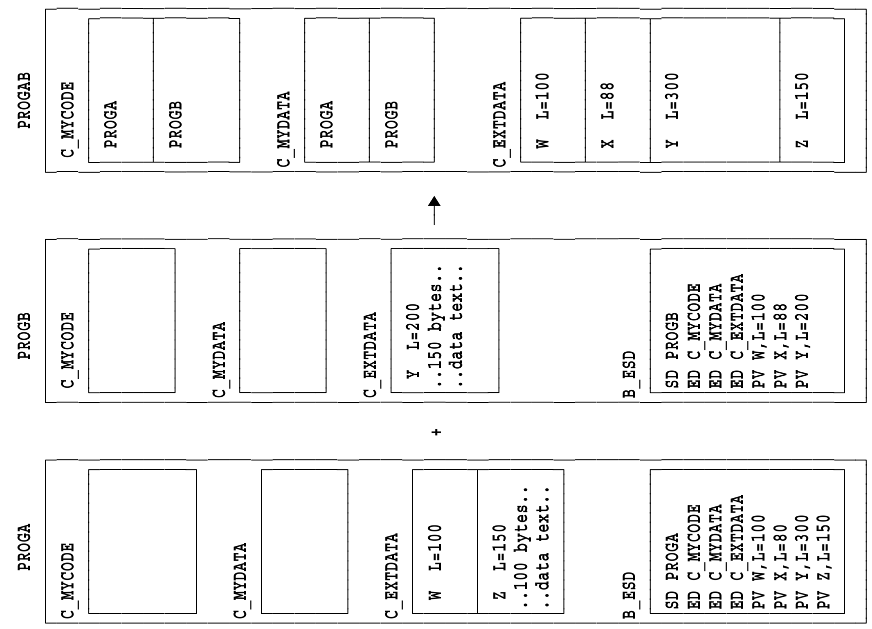
SYMBOL	DEFINED LENGTH
W	100
X	80
Y	300
Z	150

  - PROGB has defined three external data items, W, X and Y.

SYMBOL	DEFINED LENGTH
W	100
X	88
Y	200

  - If initial text was provided for W, X, Y, or Z, it would be saved in class B_PARTINIT to enable correct re-binding
- 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)

Example of External Data MERGE Binding ... 34



Part Views and Merge Binding 35

Before Binding:
Class ABCD

	Part Views			
	A	B	C	D

Sect. X	L=40
	A=3
			L=600
			A=3
				L=12
				A=0

Sect. Y	L=26
	A=2
		L=395
		A=0
				L=4
				A=2

Sect. Z	L=80
	A=3
			L=100
			A=2
				L=5
				A=0

After Binding:
Class Segment ABCD

	Part Definitions			
	A	B	C	D

Sect. X'03'	L=80
	A=3
		L=395
		A=0
			L=600
			A=3
				L=12
				A=2

Note: alignment boundary = 2^A

- PVs are “merged” by name, to determine maximum length and strictest alignment
  - Creates a ***Part Definition*** for each part name (PD is owned by a “module-level” section)
  - Parts may be ordered by priority
- PDs are then catenated within the class (in some order) to form the class's segment
- Initializing text (if any) assigned to each PD
- In this example (assuming catenation in A-B-C-D order), Class ABCD has
  - Alignment = 3 (doubleword)
  - Length = 80+395(+5)+600+12 = 1092 (the +5 aligns Part C's boundary)
- PVs kept, so that all PDs can be re-created on re-bind

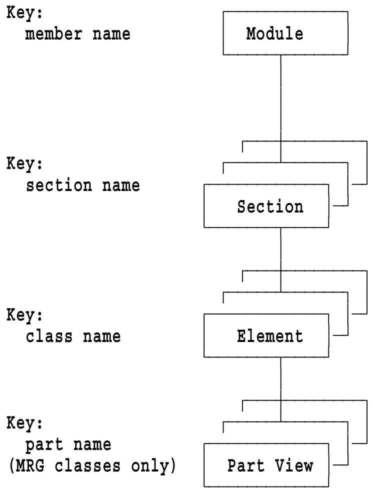
Generalized Address Constants 36

- Length of any class or part
  - Implemented in Assembler Language as J-type address constant
    - Generalization of “Cumulative External Dummy” (CXD, length of PRV)
- 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
- High order V-con bit (HOB) can be set according to AMODE of target
  - Controllable by compiler, via RLD entry
  - Controllable by bind-time “HOBSET” option, for V-cons only

Binder Inputs and Outputs

- Some pictorial views of binding and loading

Module Data: Binder Input (Logical View)

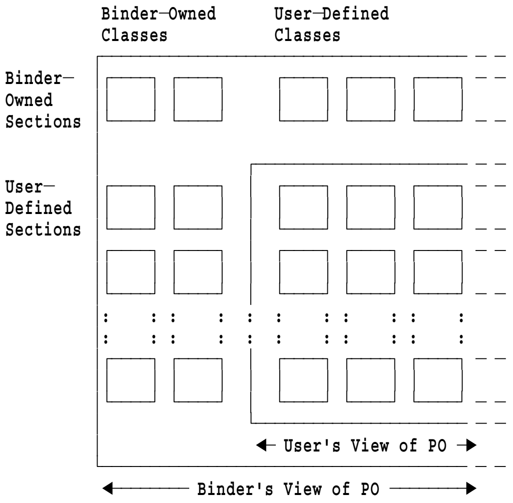


PO structure as seen by the translator and Binder user:

- ***Section*** roughly equivalent to a "compilation unit"
  - Consists of ***elements*** in various classes
- MRG classes are constructed from ***Part Views***

Binder Output view is more complex!

Module Data: Binder Output (Logical View) 39



Text classes are bound into ***segments***

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

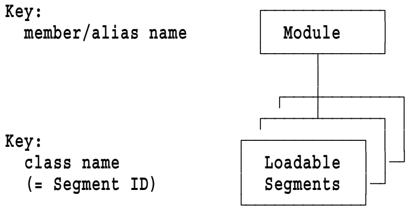
Binder retains extra “module-level” data for re-bindability

- Part Views and initializing text
- control information (e.g. B_ESD)
- IDR data, module map, etc.

in reserved section names like

- X'00000001' for B_ classes, orphaned ER/PR items
- X'00000003' for PVs, linkage descriptors, initializing data
- IEWBLIT for LE support (class B_LIT)
- IEWBCIE for DLL support (class B_IMPEXP)

Module Data: PMLoader Input (Physical View) 40



PO structure seen by PMLoader:

- PO consists of one or more ***class segments***, some of which are “loadable”
- 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

What Is a “GOFF”?

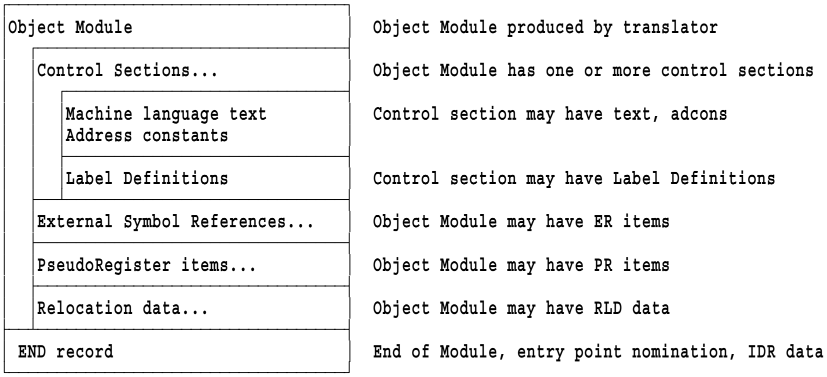
- **G**eneralized **O**bject **F**ile **F**ormat
- Complete replacement for old Object Module
  - Generated by High Level Assembler for most architected functions
  - C/C++ implementation in OS/390 V2R10
- Supports needs of languages, PO structure, Binder
  - long external names
  - 32-bit length and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

Generalized Object File Format Records 43

- 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 (OM: only on END), ADATA (OM: only in text or a side file)
  4. Relocation Dictionary: relocation information
  5. Deferred Element Length (new): formerly on OM END record
  6. End: with optional Entry-Point nomination
- Open-ended, flexible architecture; allows growth and expansion
- Documented in *OS/390 DFSMS Program Management*, SC27-0806

Structure of an Old-Format Object Module File 44

- Object modules describe a **one**-dimensional structure:

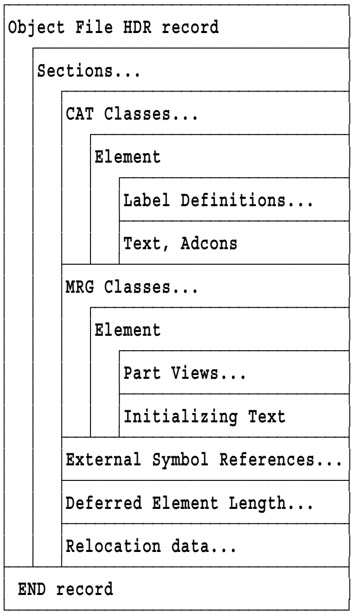


```
Object Module                         Object Module produced by translator
   Control Sections...                Object Module has one or more control sections
      Machine language text           Control section may have text, adcons
      Address constants
      Label Definitions               Control section may have Label Definitions
   External Symbol References...      Object Module may have ER items
   PseudoRegister items...            Object Module may have PR items
   Relocation data...                 Object Module may have RLD data
 END record                           End of Module, entry point nomination, IDR data
```

- Contrast with GOFF structure (slide 45)

Structure of a GOFF File 45

- GOFF files: linearized two-dimensional structure; classes in any order



Object File HDR record	Object File produced by translator
Sections...	Object File has one or more sections
CAT Classes...	Section may have CAT classes
Element	Section provides one element per class
Label Definitions...	Element may have label definitions
Text, Adcons	Element may have text, address constants
MRG Classes...	Section may have MRG classes
Element	Section provides one element per class
Part Views...	Element has one or more Part Views (no LDs)
Initializing Text	Part View may have initializing text
External Symbol References...	Section may have external references
Deferred Element Length...	Section may have deferred element lengths
Relocation data...	Section may have RLD data
END record	End of Module, may have entry point nomination

High Level Assembler GOFF-Support Options and Statements 46

- `GOFF` option creates a GOFF file
  - Existing, unmodified code will go into special “compatibility” classes
    - `B_TEXT` for text, `B_PRV` for pseudo-registers (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 14 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

High Level Assembler Support for Class Attributes 47

- Attributes currently supported by the Binder:

ALIGN(*n*) 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)

DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

High Level Assembler Support for Class Attributes ... 48

- 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 may be storage-protected

REFR The class is marked refreshable

RENT The class is marked reenterable

REUS, NOTREUS
The class is marked reusable or not

DFSMS/MVS Binder and Program Objects © IBM Corporation 1995, 2001

High Level Assembler CATTR Usage 49

- CATTR must be preceded by START, CSECT, or RSECT
  - A section name must be defined first
    - Unlike OM, no blank section is initiated
  - Text following CATTR 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

Example: A Simple Two-Class Assembler Language Program 50

- 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 name of entry point
Start    AMode 24                    Entry point has AMODE(24)
         Using *,15                  Establish addressability
Start    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
         BASSM 14,15                 Call MainCode
         — — —
         LtOrg                       RMode(24) literal pool
D31Addr  DC    A(Data31)             Addr(data above 16M)
Data24   DC    ...                   ...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)
  - Not all LDs in a section have one AMODE!

ESD From Simple Two-Class Assembly

- The assembled example creates the following 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”

Dynamic Link Libraries (DLLs)

Dynamic Linking and Dynamic Link Libraries 53

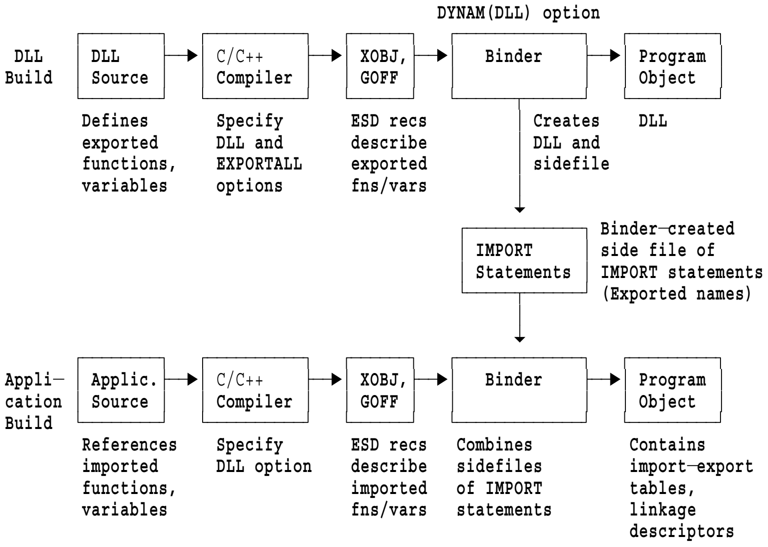
- Dynamic linking: binding of external names at execution time
  - DLLs provide one form of dynamic linking; LE is required
- DLL creator identifies names of functions and variables to be **exported**
  - Makes them available in a “side file” for runtime binding to other applications
  - Compiler indicates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

```
IMPORT  CODE|DATA,dll_name,identifier
```

  - Compilers and HLASM XATTR statement declare 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

Dynamic Linking: Preparation and Use 54

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



DYNAM(DLL) option

DLL Build

DLL Source — Defines exported functions, variables

C/C++ Compiler — Specify DLL and EXPORTALL options

XOBJ, GOFF — ESD recs describe exported fns/vars

Binder — Creates DLL and sidefile

Program Object — DLL

IMPORT Statements — Binder-created side file of IMPORT statements (Exported names)

Application Build

Applic. Source — References imported functions, variables

C/C++ Compiler — Specify DLL option

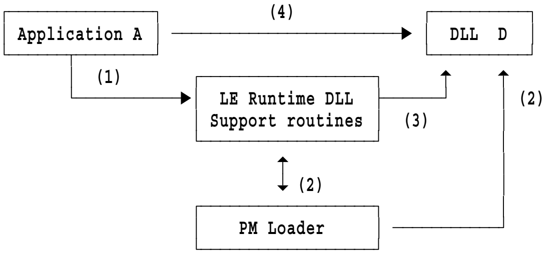
XOBJ, GOFF — ESD recs describe imported fns/vars

Binder — Combines sidefiles of IMPORT statements

Program Object — Contains import-export tables, linkage descriptors

Dynamic Linking: Execution Time 55

- Example: Application A imports names from DLL D:



Application A

(4)

DLL D

(1)

LE Runtime DLL Support routines

(3)

(2)

(2)

PM Loader

(1) First reference to an imported name passes control to LE

(2) LE DLL-support routines invoke PMLoader 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

56

Summary, Glossary, and References

Comparing Old and New 57

	Old (Load Modules)	New (Program Objects)
Components	Link Editor, Program Fetch, Batch Loader	Binder, Program Loader
Library	PDS	PDS, PDSE, HFS
Executables	One-dimensional; single RMODE	Two-dimensional; multiple segments and RMODEs
Size limit	16MB	1GB
Symbols	8 characters	1024 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

Summary 58

- 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
- Defined 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

Glossary and Definitions 59

ADATA Associated Data: program data stored in a PO which is not required for binding, loading, or execution.

API Application Programming Interface

CAT CATenate: a binding method whereby section elements within a

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.

Glossary and Definitions ... 60

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 module 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.

Glossary and Definitions ... 61

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

load module (LM)
The original form of MVS executable, stored in record format.

MRG MeRGe: a binding method whereby identically named Part Views within a class are overlaid (“merged”) before catenation.

noload A class attribute indicating that the class may be “demand loaded” by the application.

Part View (PV), PseudoRegister (PR), External Dummy (XD)
A named subdivision of a MRG class having length and alignment attributes for which space is **not** reserved in the LM or PO (see Common); used to describe a pseudoregister or external data item. Resolved to an offset within the class segment.

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.

Glossary and Definitions ... 62

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

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.

Glossary and Definitions ... 63

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 executable instructions and data. (2) A class attribute indicating that locations within the class may contain and/or be the target of address constants.

References 64

1. OS/390 DFSMS Program Management (SC27-0806)
2. DFSMS/MVS V1R3.0 Presentation Guide (GG24-4391), chapter 6
3. “Linkers and Loaders,” by Leon Presser and John R. White, *ACM Computing Surveys*, Vol. 4 No. 3, Sept. 1972, pp. 149-167.
4. Linkage Editor and Loader User's Guide
5. Linkage Editor, Loader 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)