How Link Editing and Binding Work:

Exploring Object Modules, Linking, Load Modules, and Program Objects

SHARE 112, Session 8169

John R. Ehrman
ehrman@us.ibm.com

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

March, 2009

Note: This is not a usage tutorial!

Why is this stuff important?

- Every program begins as an (un-executable) object module, and must be transformed to a loadable/executable format
- Functional limitations limit how we think about programs

What assemblers and compilers produce: object modules

Linking object modules into load modules

- Problems with load modules

How program objects are like and unlike load modules

- All about sections, classes, elements, and parts

How the new GOFF object module is like and unlike the old

How load modules and program objects are loaded into storage

Dynamic Link Library support

Glossary and references

There's a lot of material here — don't feel you have to digest it all at once....

Some frequently-used abbreviations:

PM	Program Management: the set of linking and loading programs discussed here
LM	Load Module: the traditional loadable, executable module
PO	Program Object: the new loadable, executable module
OM,OBJ	Object Module: traditional card-image format
GOFF	**G**eneralized **O**bject **F**ile **F**ormat: new format
PDS	Partitioned Data Set
PDSE	Partitioned Data Set Extended
TEXT	Machine language instructions and data
LKED	The old Linkage Editor; its functions now done by the Binder

Other terms are introduced as needed

The Way It Used to Be

(...and often still is...)

A quick review

Control section (**CSECT**) (was often called just a “section”)

- The basic indivisible unit of linking and text manipulation
- Ordinary (**CSECT**) and Read-Only (**RSECT**) have machine language text; Common (**COM**: static) and Dummy (**DSECT, DXD**: dynamic) are “templates” without text

External symbol (public; internal symbols are private)

- A name known at program linking time, whose value is intentionally not resolved at translation time; a reference or a definition

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

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
- PR names may match other external symbol names without conflict

80-byte (card-image) records, with `X'02'` in column 1,
3-character “tag” in columns 2-4

SYM Internal (“Private”) symbols (SYM records rarely used now)

ESD External Symbol Dictionary (symbols and their attributes);
each symbol (except LD) identified by an ID number: ESDID

TXT Machine language instructions and data (“Text”):
how many bytes (**Len**), where it goes (**Pos.ID, Addr**)

RLD Relocation Dictionary: data about address constants;
where it is (**Pos.ID, Addr**) and what to put in it (**Rel.ID**)

END End of object module, with **IDR** (Identification Record) data and entry-point nomination (and optional section length)

Always at least one control section per object module

One object module per compilation unit

“Batch” translations may produce multiple object modules

Describes four basic types of **external symbols**:

SD, CM **S**ection **D**efinition: the name of a control section;
CM for COMmon sections (they have no "text").
PC = Blank-named control section called "**P**rivate **C**ode";
zero-length PC sections often discarded by binder

LD **L**abel **D**efinition: the name of a position at a fixed offset within an "owning" Control Section; typically, an Entry point.
(The only type having no ESDID of its own)

ER, WX **E**xternal **R**eference: the name of a symbol defined "elsewhere" to which this module wants to refer
WX = "**W**eak E**X**ternal"; not a problem if unresolved

PR **P**seudo**R**egister: the name of a PseudoRegister
(The Assembler calls it an "E**X**ternal **D**ummy 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)

External symbol types:

SD,CM,PC Section Definition: owns LDs

LD Label Definition: entry point within an SD; no ESDID

ER,WX External Reference

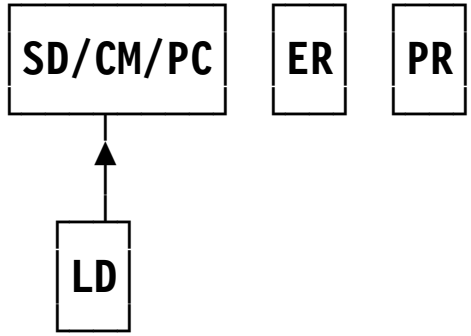
PR,XD PseudoRegister/External Dummy: this section's view of (contribution to) the PRV

Section Definition types **SQ,CQ,PQ** for quadword alignment

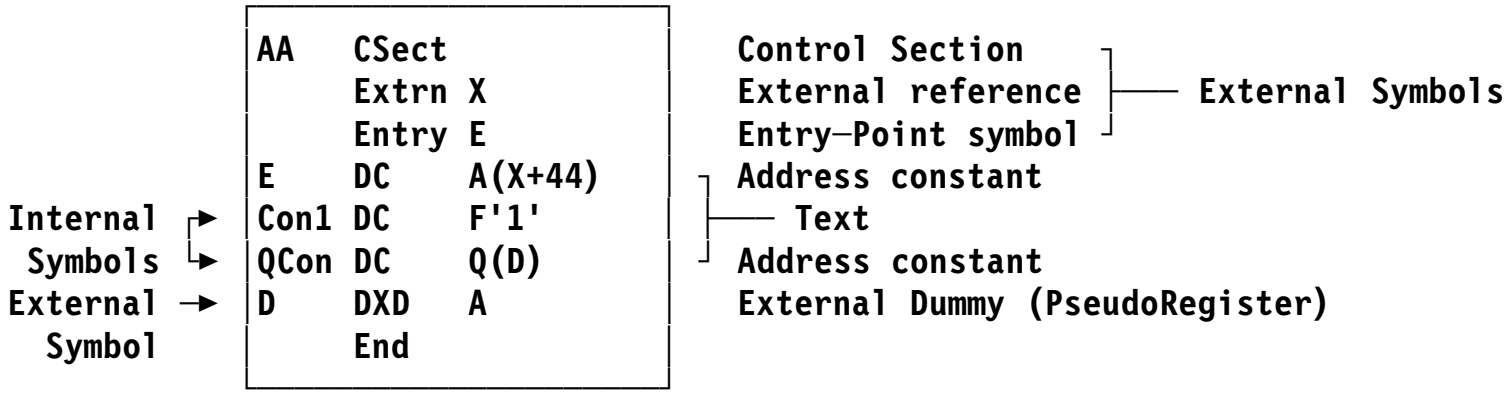
Lack of ownership of ER and PR items can sometimes cause problems when relinking

– We'll contrast this later with the new

OBJ/LM External Name Ownership Hierarchy



Sample Assembler Language program:



Assembler ESD and RLD listings:

```
                    External Symbol Dictionary
                                                    Owning ID
  Symbol   Type    ID   Addr   Length  LD ID  Flags
AA         SD   000001 000000 00000C          00    (control section)
X          ER   000002                              (EXTRN)
E          LD           000000        000001        (ENTRY in Section AA: LD ID=1)
D          XD   000003 000003 000004                (External Dummy, addr=alignment)

                    Relocation Dictionary

  Pos.Id Rel.Id   Address Type Action
000001 000002     000000   A 4   +    ...A(X): X has R–ID=2, address 0 in section AA (P–ID=1)
000001 000003     000008   Q 4  ST    ...Q(D): D has R–ID=3, address 8 in section AA (P–ID=1)
```

Combining Object Modules

A simple example of initiation-time linking

Illustrates the basic principles involved in loading and linking

Suppose a program consists of two source modules:

Module 1

```
Loc
000 MAIN
            COMMON /WORK/ ...
            – – –
            CALL SUB
            – – –
200         Addr(SUB)
204         Addr(WORK)
208         Addr(XDATA)
            – – –
            ENTRY XDATA
260 XDATA
            – – –
```

Module 2

```
Loc
000 SUB
            COMMON /WORK/ ...
            EXTERNAL XDATA
            – – –
700         Addr(WORK)
704         Addr(XDATA)
            – – –
```

(For this example, values are given in decimal)

- Program **MAIN** contains a **XDATA** entry point, and refers to the COMMON area named **WORK**
- Subprogram **SUB** refers to the external name **XDATA** and to the COMMON area named **WORK**

Translation produces two object modules

The object module for Module 1 would look roughly like this:

```
ESD SD ID=1 MAIN  Addr=000 Len=300                  SD for CSECT MAIN, ESDID=1, Len=300
ESD CM ID=2 WORK  Addr=000 Len=600                  CM for COMMON WORK, ESDID=2, Len=600
ESD LD ID=1 XDATA Addr=260                          LD for Entry XDATA, ESDID=1, Addr=260
ESD ER ID=3 SUB                                     ER for reference to SUB, ESDID=3
TXT    ID=1 Addr=000       'abcdefghijk'            Text in MAIN, address 000
TXT    ID=1 ...             etc.                    Text in MAIN
TXT    ID=1 Addr=100       'mnopqrstuvw'            Text in MAIN, address 100
TXT    ID=1 Addr=208        00000260                Text in MAIN, internal adcon offset
TXT    ID=1 Addr=260       '01234567890'            Text in MAIN, address 260
TXT    ID=1 ...             etc.                    Text in MAIN
RLD    PID=1 RID=3 Addr=200 Len=4 Type=V Dir=+      RLD item for Addr(SUB)
RLD    PID=1 RID=2 Addr=204 Len=4 Type=A Dir=+      RLD item for Addr(WORK)
RLD    PID=1 RID=1 Addr=208 Len=4 Type=A Dir=+      RLD item for Addr(XDATA)
END    Entry=MAIN                                   Module end; nominated entryname=MAIN
```

ESD records define two control sections (MAIN and WORK), one entry (XDATA), and one external reference (SUB)

RLD contains information about three address constants

- TXT for Addr(XDATA) contains offset (00000260) from MAIN

The object module for Module 2 would look roughly like this:

```
ESD SD ID=1 SUB   Addr=000 Len=800                 | SD for CSECT SUB, ESDID=1, Len=800
ESD CM ID=2 WORK  Addr=000 Len=400                 | CM for COMMON WORK, ESDID=2, Len=400
ESD ER ID=3 XDATA                                  | ER for reference to XDATA, ESDID=3
TXT    ID=1 Addr=040       'qweruiopasd'           | Text in SUB, address 040
TXT    ID=1 ...             etc.                   | Text in SUB
TXT    ID=1 Addr=180       'jklzxcvbnm'            | Text in SUB, address 180
TXT    ID=1 ...             etc.                   | Text in SUB
RLD    PID=1 RID=2 Addr=700 Len=4 Type=A Dir=+    | RLD item for Addr(WORK)
RLD    PID=1 RID=3 Addr=704 Len=4 Type=A Dir=+    | RLD item for Addr(XDATA)
END                                                | End of module
```

ESD records define two control sections (SUB and WORK) and one external reference (XDATA)

RLD contains information about two address constants

Note that both object modules start numbering ESDs at 1

The Batch Loader

1. Builds a single ("Composite") ESD to map entire program
   - Merges ESD information from the object modules; library is searched for unresolved ERs (but not WXs)
   - Renumbers ESDIDs, assigns adjusted address values to all symbols (let initial program load address be 123500)
2. Places text from SDs into storage at designated addresses
3. Determines length of COMMON (retains longest length), allocates storage for it
4. Relocates address constants by **adding** or **subtracting** relocation value to/from A-con P-field contents; by **storing** in V-cons
5. Sets entry point address and enters loaded program

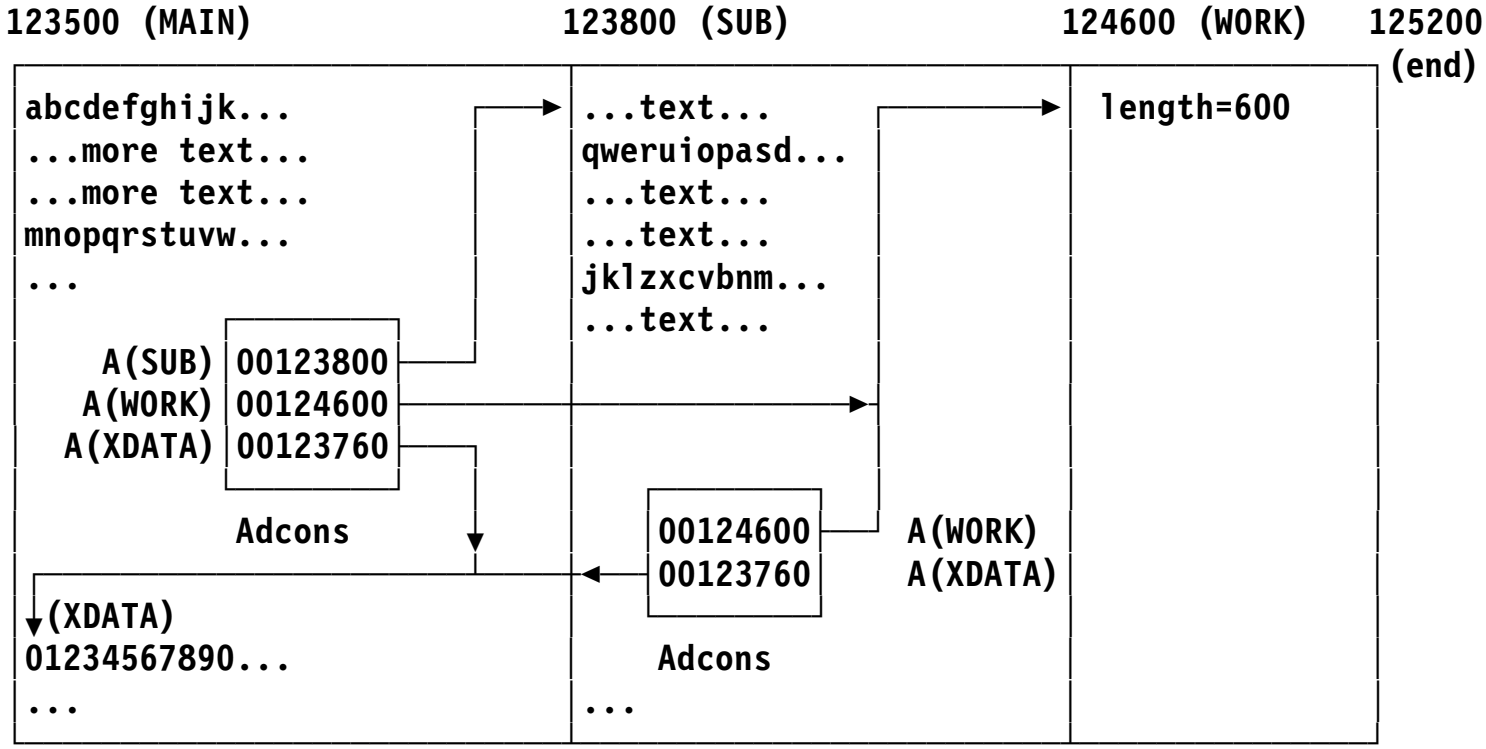
The linked program is not saved

Composite ESD (CESD)

(For this example, values are given in decimal)

Name	Type	ESDID	Addr	Length
MAIN	SD	01	123500	300
XDATA	LD	01	123760	
SUB	SD	02	123800	800
WORK	CM	03	124600	600
(end)			125200	
entry		01	123500	

The resulting program, loaded into storage for execution:



Storage was allocated for three control sections (two SD, one CM)

Address constants were resolved to designated addresses

Loader enters program at entry point MAIN (123500)

Saving Linked Programs: Load Modules

Same linking process as in previous example, except:

- Assumed “origin address” for load modules is zero
- Program written to DASD
- Unresolved ERs OK if NCAL option is specified
- Final relocation will be done by the Program Loader

Load module's structure very similar to object module's

- Simplifies processing of each

Basic contents (analogous to object module records)

SYM	Object-module records copied directly into load modules
IDR	Identification records (from object module END records, Linkage Editor, User, and SuperZAP)
CESD	Composite External Symbol Dictionary
TEXT	Machine language instructions and data
RLD	Relocation Dictionary (in control records)
EOM	End of module (a flag field in a control record)

Additional items having no object-module analogs

CTL Control records, for reading and relocating text records

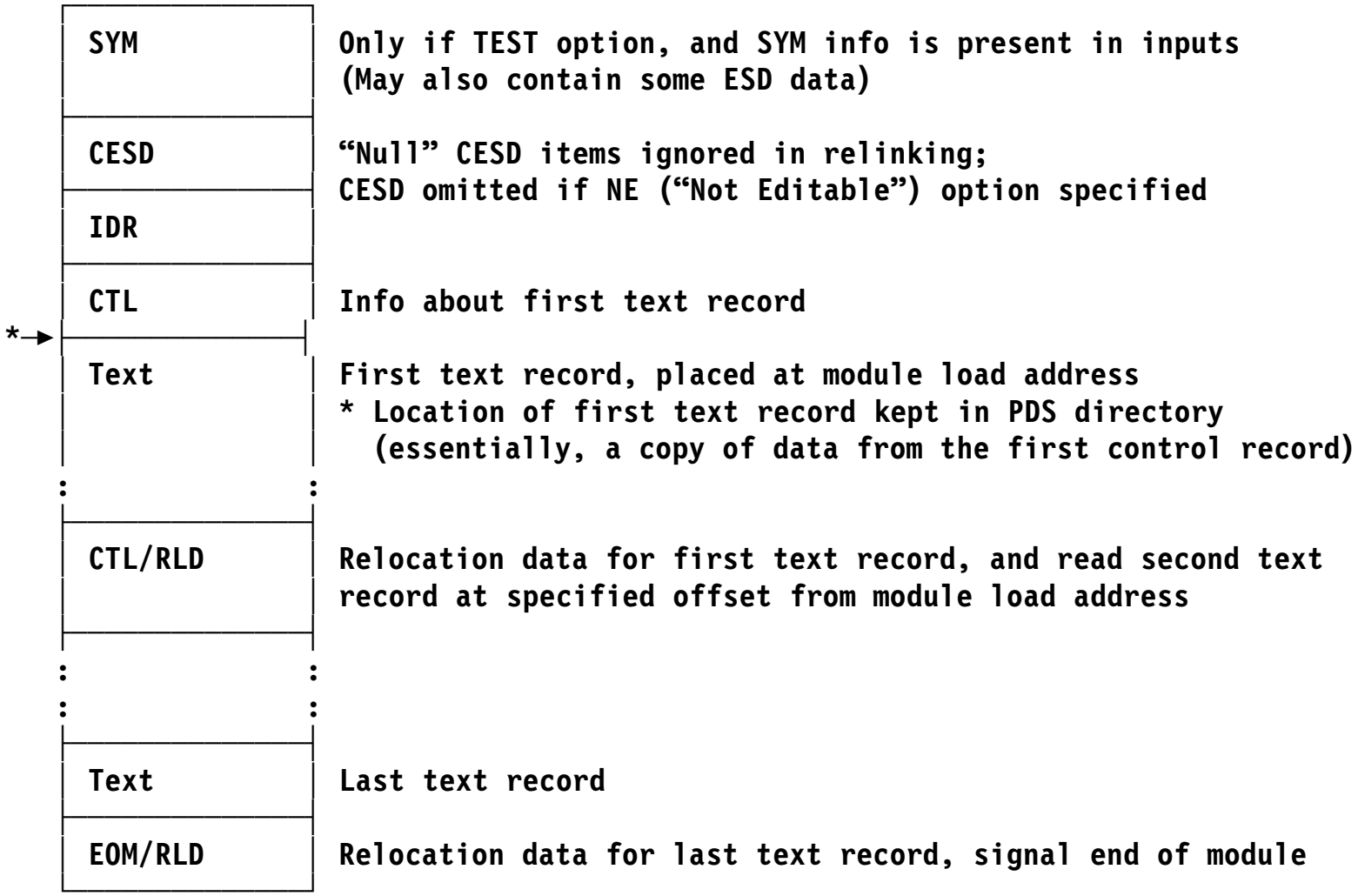
And, for modules in overlay format:

SEGTAB Segment table

ENTAB Entry table

EOS End of Segment (a flag field on a CTL record)

Basic format called "record format," "block format" or "block loaded"



PseudoRegister Processing

PseudoRegisters not used frequently today

- Originally used in OS/360 for reentrant PL/I applications

Their binding principles are heavily used in POs

- Especially for shared and private work areas in C/C++ applications

Allow sharing *by name* of dynamically managed external objects defined in separately translated re-entrant programs

- Originally required by OS PL/I for files, areas, controlled variables, etc.

PRs have their own “name space”

- Separate from all other external symbols
  - PR names may be identical to other types of ESD name without collision

PR items refer to offsets in a “link-time Dummy Control Section”

- A template; a data-structure mapping created at link time
- Hence the Assembler's name, “External Dummy” (XD)
- The dummy section is also called a “PseudoRegister Vector” (PRV) (PL/I's PRV allowed up to 1024 more 32-bit “registers”)
- All names are known and bound at link time (Example on slide 22)

A more generalized form (a “Part”) is used in program objects

- Binding similar to PseudoRegisters

PR's are resolved somewhat like commons,
but no storage is allocated at link time

- For multiple definitions, longest length and strictest alignments win
- Accumulated length/alignment of PRV items then determine offset associated with each PR name
- Offset value placed in Q-type address constants referencing PR name
- Total size of the “link-time DSECT” (up to 2GB) is placed in “CXD” adcon items

PR and CXD resolution is completed at link time

Runtime code acquires a storage area of the CXD-specified size

Runtime references access fields at desired offsets into the acquired area

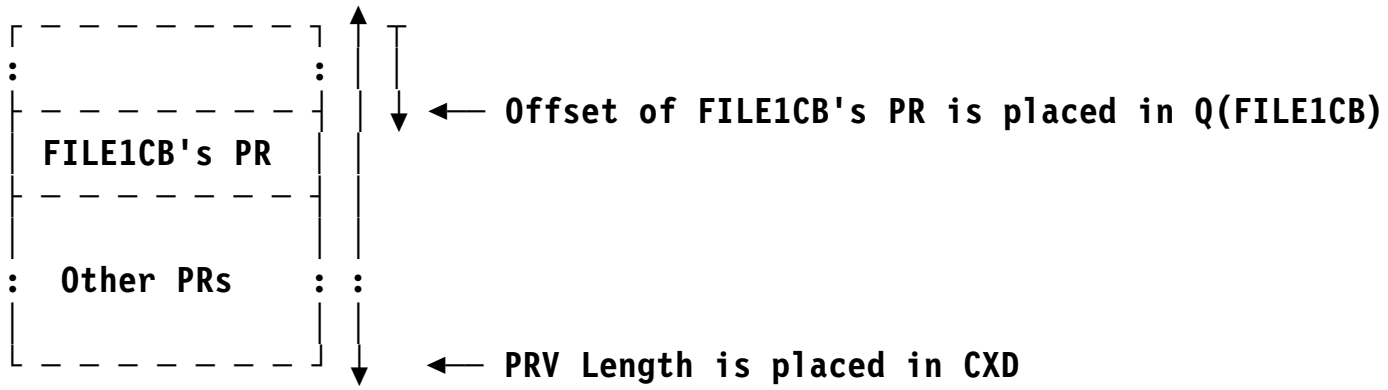
- Q-con contents provide displacements

The following example illustrates this process

Declare XD/PR for “FILE1CB” in each referencing program:

```
FILE1CB  DXD    A              Will hold address of File 1's Control Block
```

Link with other modules; Binder creates “virtual” PRV



Each invocation of the main program acquires storage for its real PRV:

```
         L    0,PRVLen       Get length of PRV
         GetMain R,LV=(0)    Get storage
         LR   11,1           Carry PRV address in R11
         - - -               ...initialize contents appropriately...
PRVLen   CXD                 Link Editor inserts total length of PRV
```

Modules reference PRV's FILE1CB field using offsets in Q-type adcons:

```
         L    2,=Q(FILE1CB)  Get PRV offset of FILE1CB pointer
         AR   2,11           Storage address of FILE1CB pointer
         L    1,0(,2)        Pointer to FILE1CB now in R1
```

COMMONs and PseudoRegisters have similarities and differences

	COMMONs	**PseudoRegisters**
Bind-time behavior	Space allocated in the load module	No space allocated; a mapping of all PR items into a virtual PRV
Storage Allocation	Static: part of the load module	Dynamic: at run time
Initialization	None	Run-time code's responsibility
Copies	One per load module	One per load module; instantiated during each execution
External names	One per common, one per load module	One per PR; no conflict with non-PR names
Internal names	As many as you want	None (unless you map the PR's inner structure with a DSECT)
References	Direct, with adcons	One level of indirection via Q-con offsets and the base reg anchoring the allocated storage

Overlay Modules

Not much used today, but worth knowing about

- They provide some potentially useful capabilities

Overlay modules share the same storage (at different times!)

Suppose MAIN calls SUBA and SUBB

- Neither SUB calls the other

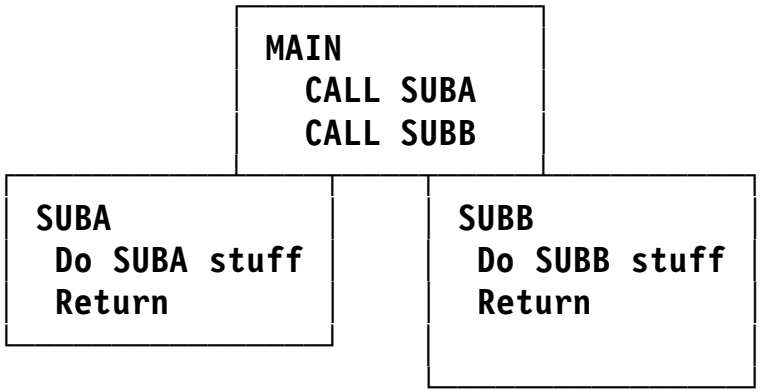
In block format, they would appear in storage as

```
MAIN
  CALL SUBA
  CALL SUBB

SUBA
  Do SUBA stuff
  Return

SUBB
  Do SUBB stuff
  Return
```

SUBA and SUBB might be overlaid, like this:



SUBA and SUBB share the same storage

The overlay supervisor must (help) make this work!

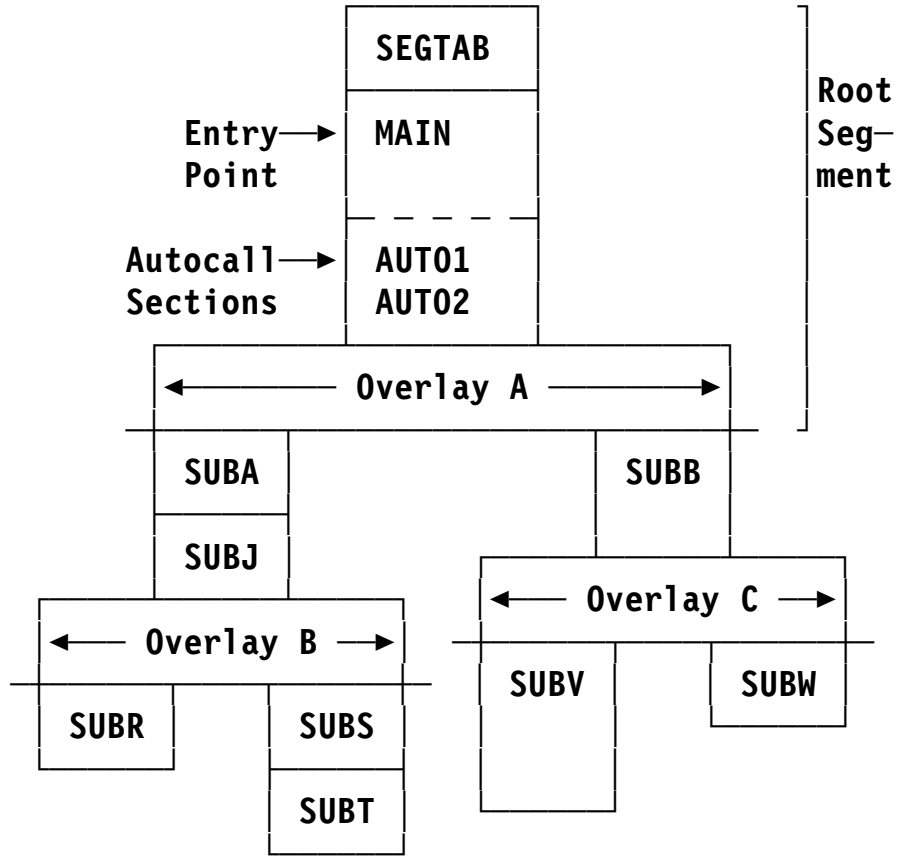
Determine how modules can share storage

Draw an “overlay tree” of the structure

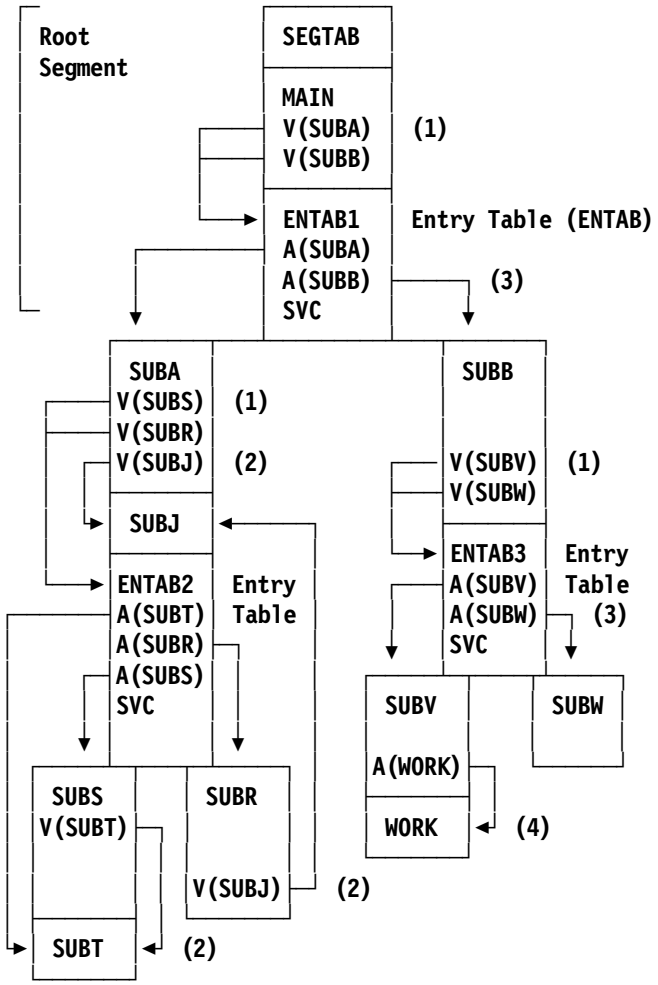
- Root (low address) at top
- Control statements describe desired structure
- In this example, three overlay nodes: A, B, C; seven loadable segments

Root segment is **always** present

- Contains entry point, autocalled sections, Segment Table (SEGTAB tells what segments are in storage)



An Overlay Structure In More Detail



Each segment with subsidiary segments is suffixed with an **Entry Table** to assist loading of the “lower” segments

- SVCs call Overlay Supervisor

V-type adcons may resolve to an ENTAB, not to the named symbol!

- V-cons for SUBs in lower segments resolve to ENTAB **(1)**
- V-con for call in same or higher segment resolves directly **(2)**

A-cons always resolve directly

- In ENTAB, resolve directly to SUBs **(3)**
- To sections in same segment **(4)**
- Block format may work, but not overlay!

Segment reload resets local data!

- Which may be good or bad!

Overlay Regions

Overlays can be arranged in independent groups: REGIONs

- Allows greater freedom in structuring programs

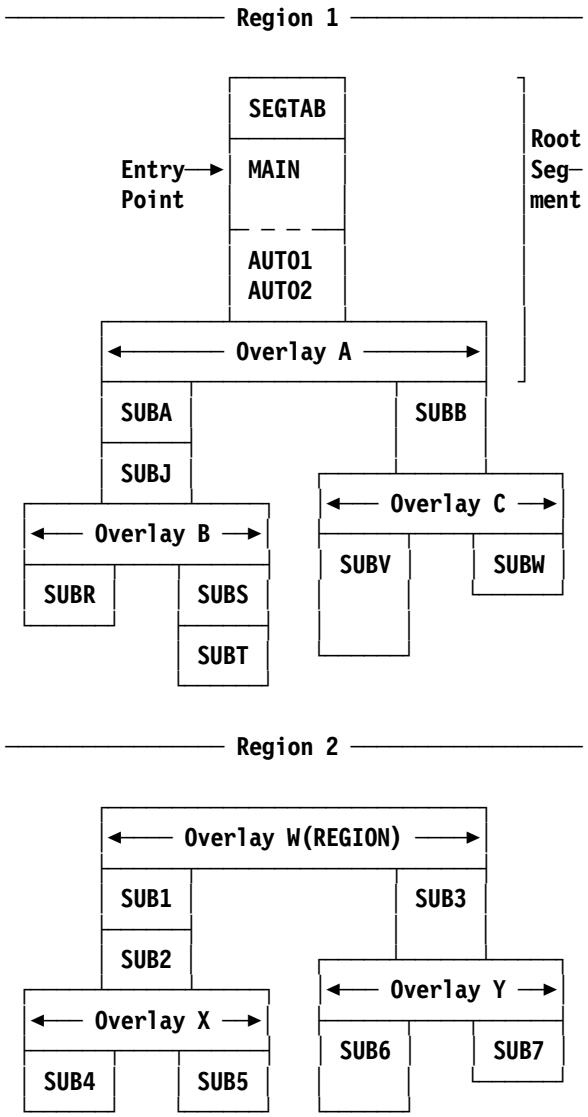
Each region can be a separate overlay structure!

- Up to four regions

A form of dynamic loading

- Specific routines loaded as needed
- No displacing of segments in other regions

Example with two regions:



Pro:

- Faster initiation: only part of the program need be loaded to start
- Economical storage use: only load what's needed, when it's needed
- Modules can handle **more** than 16M of text
- Can always re-link to block format if there's enough storage
  - ***But:*** Behavior may be different, due to loss of re-initializations!
- Can help applications grow if dependent on 24-bit addressing

Con:

- AMODE, RMODE must be 24
- Programs are not re-enterable, cannot be shared
- More complex to specify; greater care needed in coding certain items:
  1. Local data may or may not “persist” across calls
  2. External data sharing protocols may be more complicated
  3. V-type adcon references may be indirect! (A-type is always “direct”)
- Additional overhead in calls to segments needing to be loaded
- Calls among certain modules may be forbidden (or wrong)

Problems with Load Modules

(...despite their incredible durability...)

Early-binding philosophy: systems are expensive, people are cheap

- Programs run for long periods between needed changes
- Therefore: recompile “deltas” and re-link them into the application module

Re-linking is much cheaper than re-building modules “from scratch”

- Therefore: keep enough info with the module to simplify re-editing

DASD is slow, and central storage is precious and very expensive

- Therefore: short records are a good thing
- Therefore: packing module pieces tightly is a good thing
- Therefore: overlay structures are very good things

24-bit addresses and lengths are adequate for a very long time

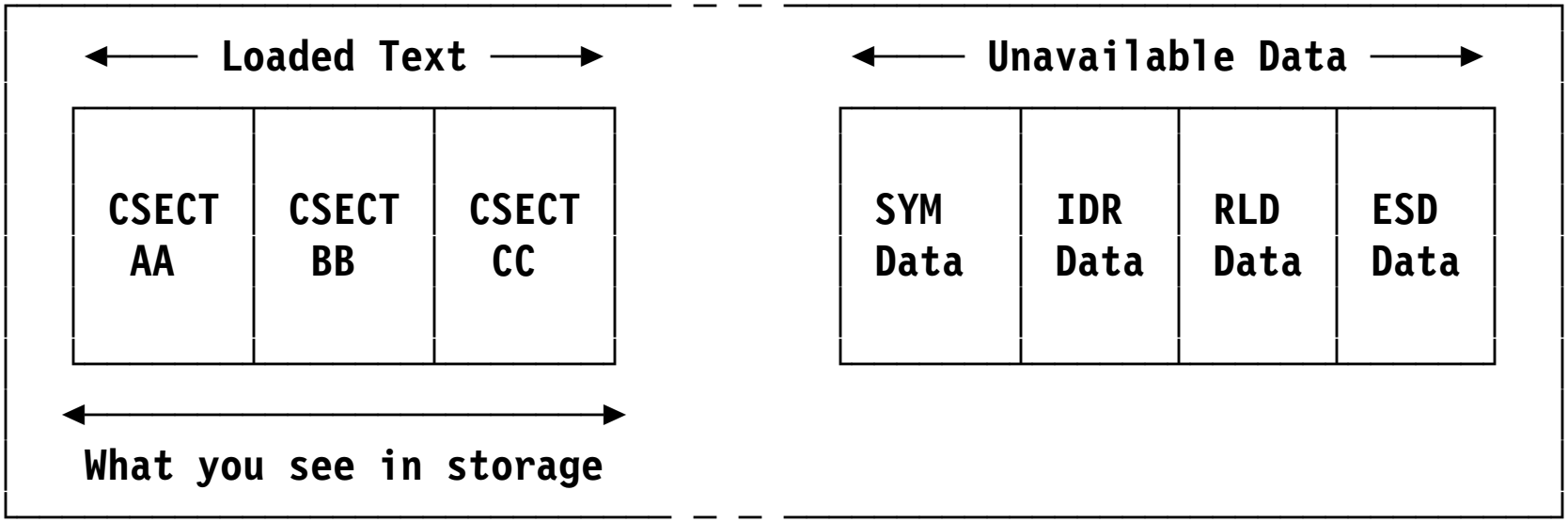
- Therefore: Everything must be smaller than 16MB
- Therefore: AMODE and RMODE were “patched in” later
- Therefore: no “scatter loading” by RMODE; entry points don't have own AMODE

8-character upper-case EBCDIC names (vs. BCD's 6) are adequate

Central storage is real (not virtual)

- No page-outs of relocated pages (a problem with block-loaded LMs)

Load modules have a one-dimensional 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 “unavailable” module data not accessible via normal services (prior to implementation of Binder APIs)

Gap: any area of load module text not specified by inputs

- Explicit request (such as assembler's DS statement)
- Areas skipped for alignment (within sections, ends of sections)
- Uninitialized COMMON areas

Gas: LMs may contain short text-record blocks; PDS has dead modules

- Large gaps: write out the just-completed text record
  - Also depends on space left on track (impenetrable algorithm decides)
- Only one partial CSECT allowed per block
- Too many dead/replaced modules? PDS compression required!

Initial values: what eventually appears in the gaps?

- Small gaps: depended on what was in the Binder/Link Editor text buffers
  - In early days, could be anything (now cleared by default to zeros)
- Large gaps: may depend on what's in storage during module loading

Binder's FILL option lets you specify a value (helps find uninitialized variables)

POs and PDSEs solve many of these problems

Advice: never depend on anything you didn't initialize

- FILL option and storage allocation may or may not initialize to zero!

SYM and IDR put at front of module, to simplify Link Editor logic

CESD is at front of module, to simplify re-processing of load modules

PDS directory info allows Program Fetch to skip this stuff

- First text record's length and disk location; storage needed; attributes; etc.

Small record lengths

- SYM ≤ 244; CESD ≤ 248; IDR, CTL, RLD ≤ 256; Text ≤ track length
  (Linking can force text records to be much shorter than a track)

If first “real” text is not at relative zero, write a 1-byte record at zero!

- Required by Program Fetch, Program Loader

“Directory name space” (PDS directory names) independent of external (CESD) names (which can be independent of internal names, too!)

- Can assign member and alias names unrelated to CESD names
  - Member MM creates an object module containing symbol AA, which is renamed during linking to BB, stored in PDS member CC

Format is externalized! (And therefore unchangeable...)

Program Objects

Format is not externalized

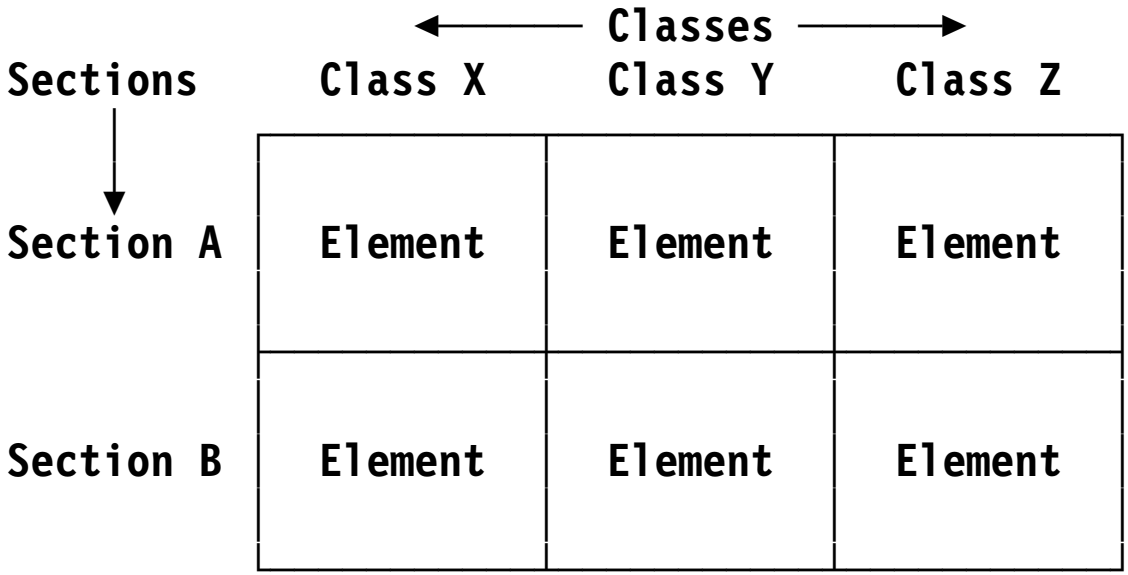
- Has already taken several different formats as requirements evolved
- Each new format extends PO's capabilities
- All information available via Binder APIs
  - Full-function and “FAST DATA” (for read-only access)

Similar linking process as for load modules, except:

- Program written to a PDSE or UNIX file (not to a PDS)
- Final relocation is done by Program Loader

Some new terminology; some old terms are used differently

Most easily visualized as a two-dimensional structure:



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!)

The unit defined by a section name and a class name is an ***Element***

A PO section is a “handle” or a “cross-section”

- Neither a CSECT name nor an external name
  - There are no “Control Sections” in a PO!
  - Traditional CSECTs are mapped to elements

Each section supplies element contributions to one or more classes

- According to their desired binding and loading characteristics

Section names must be unique within a Program Object

- As for Load Modules
- Section names are not external names or implied labels; not used to resolve external references

A section is the program unit manipulated (replaced, deleted, ordered, or aligned) during binding

- Operations on a section apply to all elements within the section
- Including rejection; only the first occurrence of a section is kept

Class

- Each PO class has uniform loading/binding attributes and behavior
- Attributes assigned when the class is defined in your program
  - Most important: RMODE, Loadability, Text type
- Several classes may have identical attributes
- Elements in each class are bound together in a ***Segment***
- Class names (max. 16 characters) rarely externalized or referenced
  - Names of the form **`letter_symbol`** are reserved to IBM

Element

- The indivisible unit of text (analogous to OM/LM CSECT)
- Contains machine language instructions and data
- Not named; identified by owning Section and Class
  - **Note**: Binder listings describe an element as a “CSECT”
- Label Definitions (LDs) within elements identify positions in text

Segment: a bound set of elements with the same binding/loading attributes

- Binder may combine classes with identical attributes into one segment

Class loading attributes determine the load-time placement of segments in virtual storage

- Most important attributes are RMode and Loadability
- Not all segments are loadable; depends on attributes

Loadable segments are loaded as separately relocated discontiguous entities

- Loaded as a single “block”; similar to a load module
- Inter-segment references are correctly resolved

Multiple attributes may be assigned to each class, such as:

RMODE: indicates placement in virtual storage of a loaded segment

Text type: byte-stream (machine language) or record-like (IDR, ADATA)

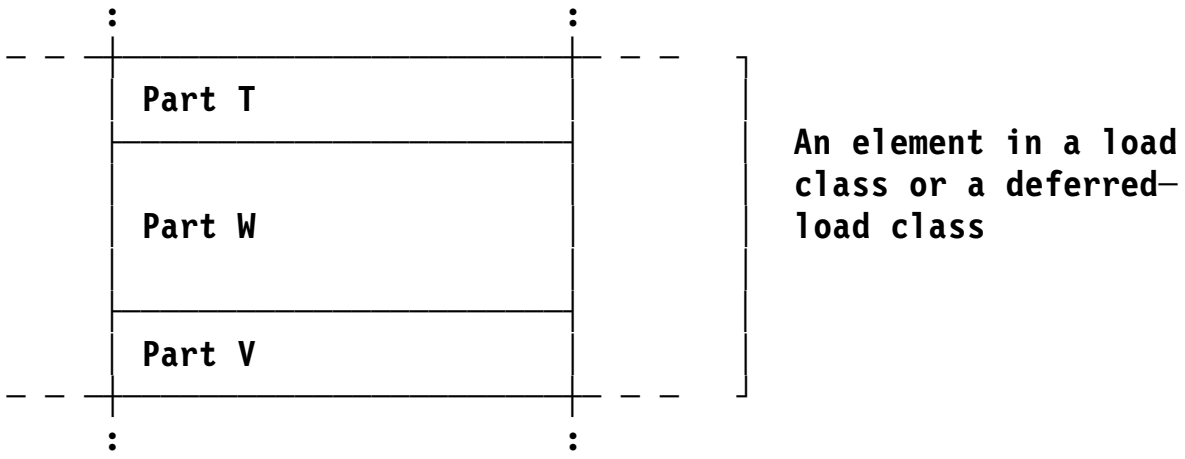
Loadability

- **LOAD**: The class is brought into memory when the program is initially loaded
  - Same as load module's usual behavior
- **DEFERRED LOAD**: The class is prepared for loading, and is instantiated when requested
  - RENT applications typically contain non-RENT deferred-load classes
  - For data such as pre-initialized private writable static data areas in shared (re-entrant) programs (in Assembler, “PSECTs”)
- **NOLOAD**: The class is not loaded with the program; may not contain adcons
  - May contain any useful data to be kept with the program
  - Non-text classes are always NOLOAD; application access via Binder APIs

Boundary alignment

Part: a component of class with the MERGE attribute

- Multiple Parts per element allowed
- May be in a Load or a Deferred-Load class



Often used as a template for *static* read/write component of RENT applications

- Typically: external-data definitions, local variables, code fragments, adcons referencing other parts, linkage descriptors, constructors/destructors

Classes have one of two binding attributes: Catenate, Merge

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

2. ***Merge*** (MRG) (The name is misleading)

   A generalization of Linkage Editor and Binder handling of CM, PR items

   Section contributions to MRG classes: Commons, PseudoRegisters, Parts

   - Parts may contain text (unlike Commons and PseudoRegisters)

   First appearance of a Part determines length, alignment; others are rejected

   Parts within a section and class are catenated to form an element

   - Different from CAT binding: the element is built by the Binder

   Typically used for classes that must group small, related items together

   - Each section supplies any number of data items

Five external symbol types:

SD ***Section Definition***: owns other types

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

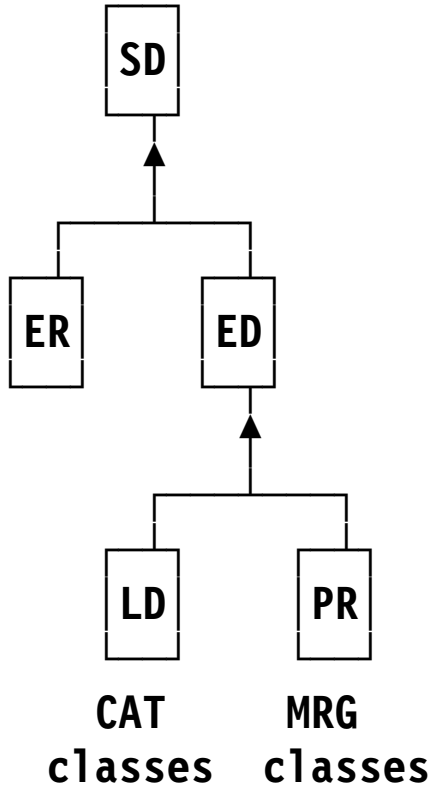
LD ***Label Definition***: entry point within an element; owned by an ED; only in CAT classes; has own ESDID and AMODE (unlike OBJ)

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, as noted on slide 7)

New External Name Ownership Hierarchy



Six record types (similar to the five OBJ types)

HDR Module Header (new): CCSID, translator identification, etc.

ESD External Symbol Dictionary: long names; symbol types and attributes; 64-bit address/offset fields; multiple classes

TXT Text: machine language object code, IDR, ADATA

- OBJ: IDR only on END; ADATA only in text or a side file

RLD Relocation Dictionary: relocation information

LEN Deferred Element Length (new)

- In case anyone still uses this old OBJ END-record function
- No known current uses; provided only for compatibility

END With optional entry-point nomination

No SYM record; internal symbol data usually in an ADATA class

AMODE assignable to entry points

Four symbol **scopes**:

Section New; symbols resolved only within the section

Module Same as Weak External (WX): no library search if unresolved

Library Same as Strong External (ER): library search if unresolved

Import-Export
New; symbols resolved during execution (see slides 65-67)

Open-ended, flexible architecture; has grown and expanded as needed

- Variable-length or FB80 records

Sample program, assembled two ways:

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

MyCom  COM   ,        (CM)
       DS    12D

Sect_B CSect ,        (SD)

My_XD  DXD   3D       (XD)

       Entry B_Data   (LD)
B_Data DC    7D'1.0'

       End   Sect_A
```

GOFF: CSect mapped to SD/ED, CSect name now an LD

OM ESD (HLASM **OBJECT,NOGOFF** option)

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

GOFF ESD (HLASM **OBJECT,GOFF** option)

```
Symbol   Type   Id     Address  Length   LD ID
SECT_A    SD 00000001
B_IDRL    ED 00000002                    00000001 (new)
B_PRV     ED 00000003                    00000001 (new)
B_TEXT    ED 00000004 00000000 0000002C 00000001 (new)
SECT_A    LD 00000005 00000000           00000004 (new)
MYCOM     SD 00000006
B_IDRL    ED 00000007                    00000006 (new)
B_PRV     ED 00000008                    00000006 (new)
B_TEXT    ED 00000009 00000000 00000060 00000006 (new)
MYCOM     CM 0000000A 00000000           00000009
SECT_B    SD 0000000B
B_IDRL    ED 0000000C                    0000000B (new)
B_PRV     ED 0000000D                    0000000B (new)
B_TEXT    ED 0000000E 00000000 00000038 0000000B (new)
SECT_B    LD 0000000F 00000000           0000000E (new)
MY_XD     XD 00000010 00000007 00000018
B_DATA    LD 00000011 00000000           0000000E
```

– HLASM generates B_IDRL, B_PRV and B_TEXT class definitions and LD for each SD item

Define two sections and two classes, one with parts

```
Sect_A  CSect ,                 Define first section, 'Sect_A'
*
Sect_A  RMode Any               RMode inherited by default element
        DC    A(Sect_B)         Address of default element in 2nd section
*
Class_X CAttr RMode(24)         Class 'Class_X' declared
        BR    14                Do something?
*
Sect_B  CSect ,                 Define second section, 'Sect_B'
        DC    A(Sect_A)         Address of default element in 1st section
*
Class_X CAttr ,                 Contribution to 'Class_X'
        NOPR  0                 Do nothing?
*
Class_Y CAttr RMode(31),Part(T)    Define Part T in 'Class_Y'
***************************** Part(T) means 'Class_Y' is a MRG class
        DS    D                 Working storage
*
Class_Y CAttr Part(W)           Define Part W in 'Class_Y'
        DC    A(T)              Address of Part T
*
Sect_B  CSect ,                 Resume default element in 'Sect_B'
        BR    14                Now do something?
        End
```

```
                                External Symbol Dictionary

Symbol  Type   Id     Address  Length  Owner Id Flags Alias-of

SECT_A   SD 00000001                                  <-- SD
B_IDRL   ED 00000002                   00000001       <-- Generated ED
B_PRV    ED 00000003                   00000001       <-- Generated ED
B_TEXT   ED 00000004 00000000 00000004 00000001    06 <-- Generated ED
SECT_A   LD 00000005 00000000          00000004    06 <-- Generated LD

CLAS_X   ED 00000006 00000008 00000004 00000001       <-- Declared ED

SECT_B   SD 00000007                                  <-- SD
B_IDRL   ED 00000008                   00000007
B_PRV    ED 00000009                   00000007
B_TEXT   ED 0000000A 00000010 00000006 00000007    00
SECT_B   LD 0000000B 00000010          0000000A    00 <-- Generated LD

CLAS_Y   ED 0000000C 00000000 00000000 00000007       <-- Declared ED

T        PD 0000000D 00000000 00000008 0000000C    06 <-- Part T
W        PD 0000000E 00000000 00000004 0000000C    06 <-- Part W

                                    Relocation Dictionary
   Pos.Id   Rel.Id   Address  Type  Action
  00000004 0000000A 00000000   A 4     +
  0000000A 00000004 00000010   A 4     +
  0000000E 0000000D 00000000   A 4     +
```

Binder-created sections contain module-level data

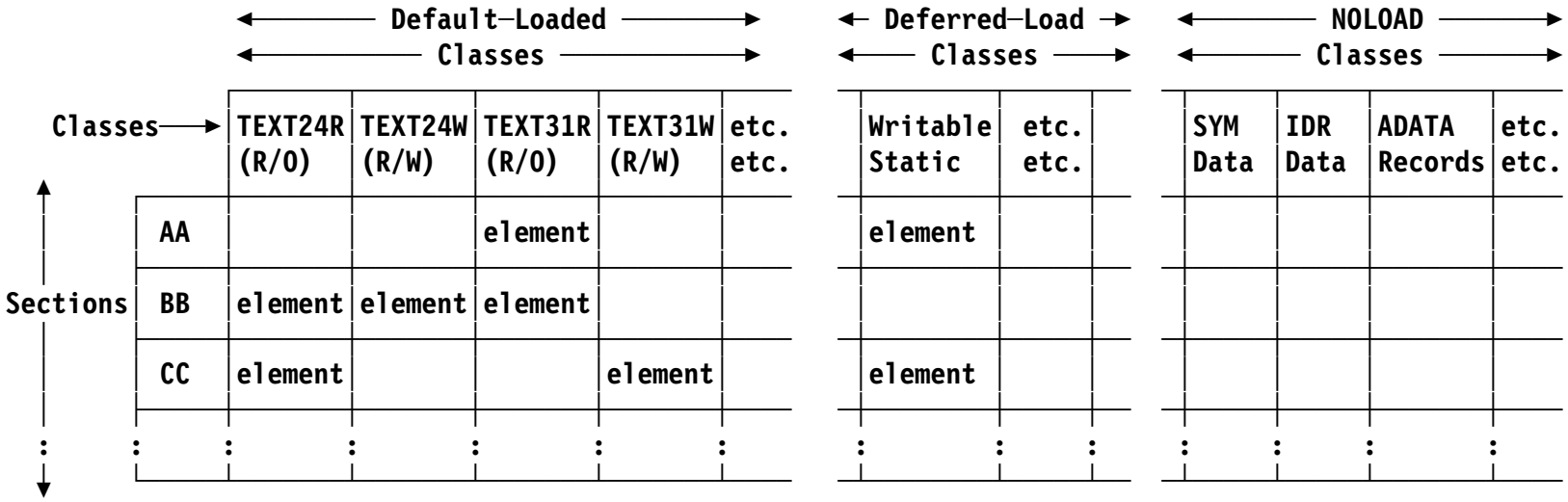
- ESD data, class maps, SYM data, module-level ADATA, Part Definitions
- Avoid section and external names starting with **IEWB** (see slide 61)

Binder-created classes contain data needed for correct re-binding

- Example: names like **C_xxx** reserved to LE and compilers, **B_xxx** to Binder

  B_ESD contains external names

  B_IMPEXP contains imported/exported external names (for DLL support)



All elements in a 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 32)

All classes (including NOLOAD) accessible via Binder APIs

Deferred-load classes require special Program Loader interface

- Currently, only a single DEFLOAD class is supported (and only by LE)

Compatibility of Old and New

All functionality of old OM/LM behavior is retained

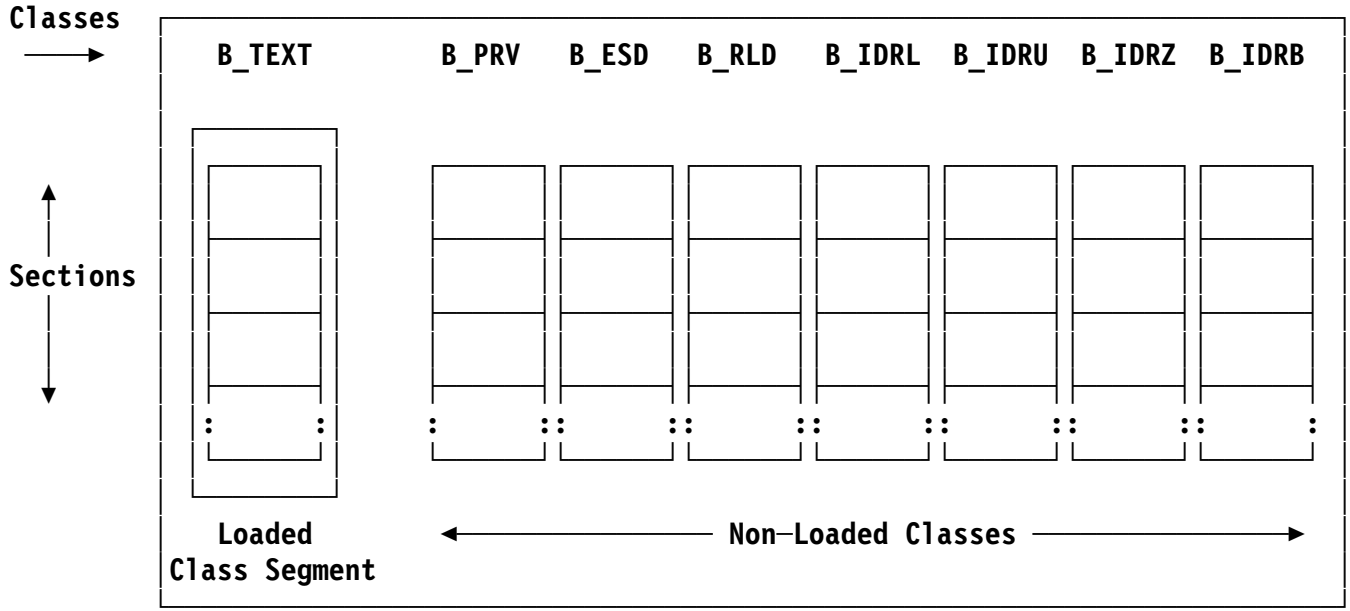
Old object code is mapped by the Binder:

OM/LM	Binder's Program-Object 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 SD name to each (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

Using IEBCOPY to convert LM (PDS) to PO (PDSE) invokes the Binder

Old load 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 **L**anguage translators (L), **U**ser (U), Super **Z**ap (Z), and **B**inder (B)

Link Editor: linking modules with mixed RMODEs forces the LM to most restrictive value

- Old way to create programs with 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)
  - Move chunks of code above or below the line

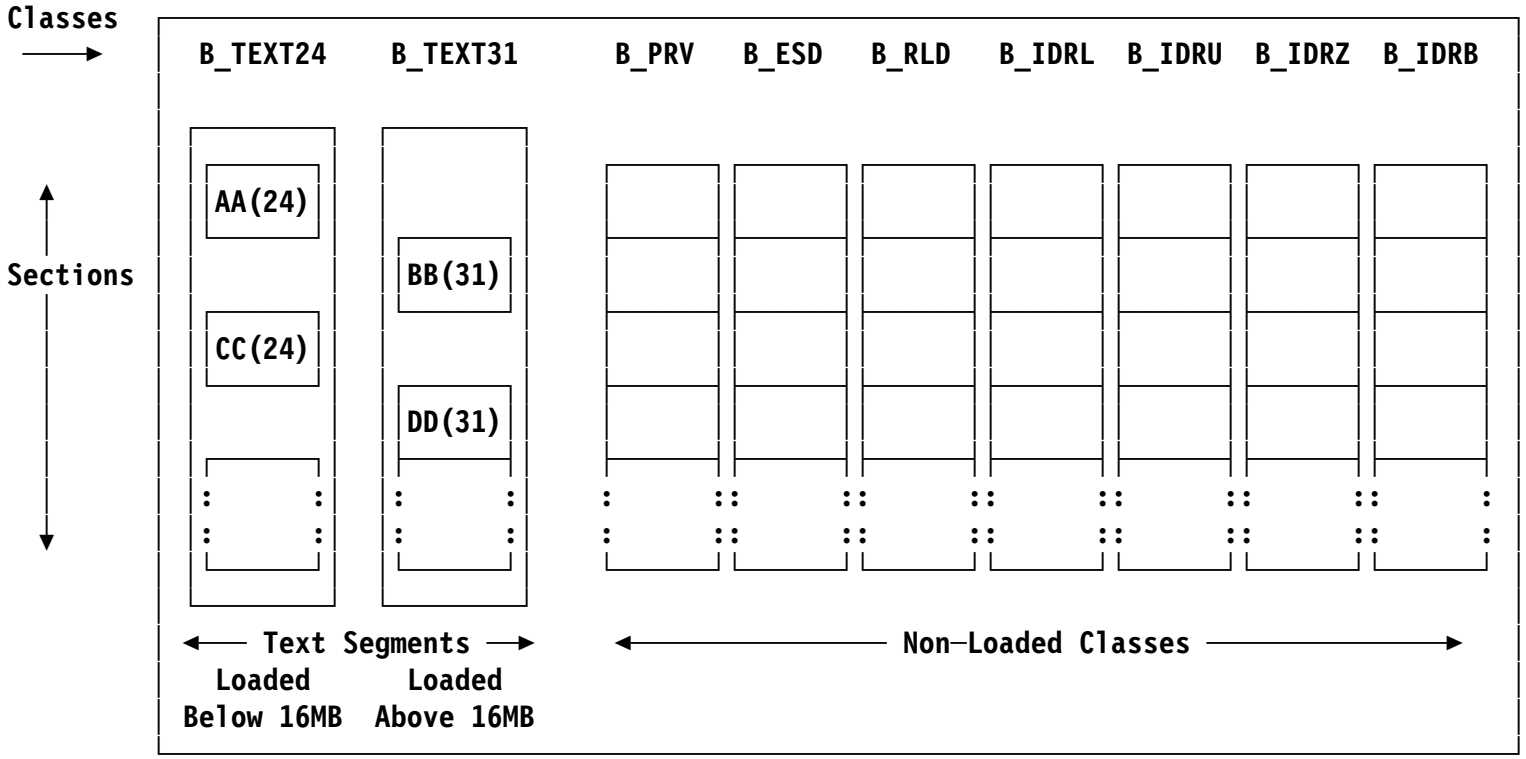
Binder: **RMODE(SPLIT)** option creates a PO with two text classes

- Affects only class B_TEXT:
  - RMODE(24) CSECTs (from class B_TEXT) moved to B_TEXT24 class, RMODE(31) CSECTs (from class B_TEXT) moved to B_TEXT31 class
  - B_TEXT24 class loaded below 16M, B_TEXT31 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!
- 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 RMode(24) and RMode(31) classes

– Compare to slide 53



Inter-class references resolved automatically

Easiest if program runs uniformly in AMode(31) (if possible!)

Loading Modules into Storage

Program Fetch

- Traditional LM loader

Program Loader

- New PO/LM loader
- Includes all functions of Program Fetch

Sequential I/O for loading all load modules (LOAD, LINK, XCTL, ...)

Skip over everything preceding the first control record

- SYM, IDR, CESD (PDS directory info makes the skipping simple; see slide 18)

Control records tell length, relative address of next record's text

- May also contain RLD information for preceding text block

A,V-cons relocated using only **address** information in RLD,
and only by adding the module's load address

- Only a **single** relocation base
- Q-cons and CXDs were completed at linkage-edit time

Two stages of relocation are involved:

1. LKED, Binder: relocate addresses relative to zero module origin
2. Program Loader: relocate addresses relative to module's “load address”

Overlay Supervisor

- SEGTAB and ENTABs manage segment traffic; Program Fetch loads segments as requested

Each **segment** relocated independently

"Linear" format uses efficient "DIV" mapping to virtual storage

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 UNIX files
  - — Under z/OS Unix Services, POs in UNIX files are written/read as "flat 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:

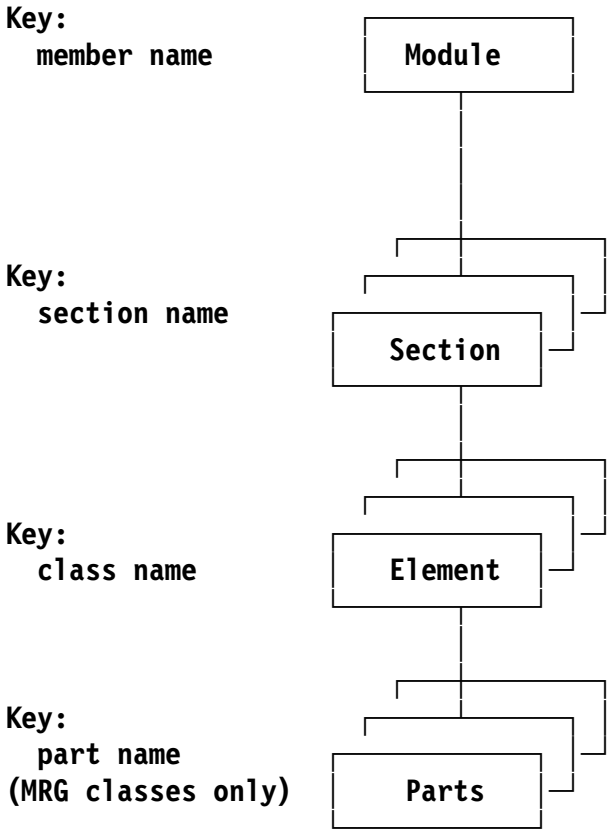
- Preloads and maps entire module in intermediate storage, then moves to destination

Supports RMODE(64) loading "above the bar"

- Use this **only** for data!
- Currently supported only for LE's **C_WSA64** deferred-load class

Binder Inputs and Outputs

Some pictorial views of binding and loading



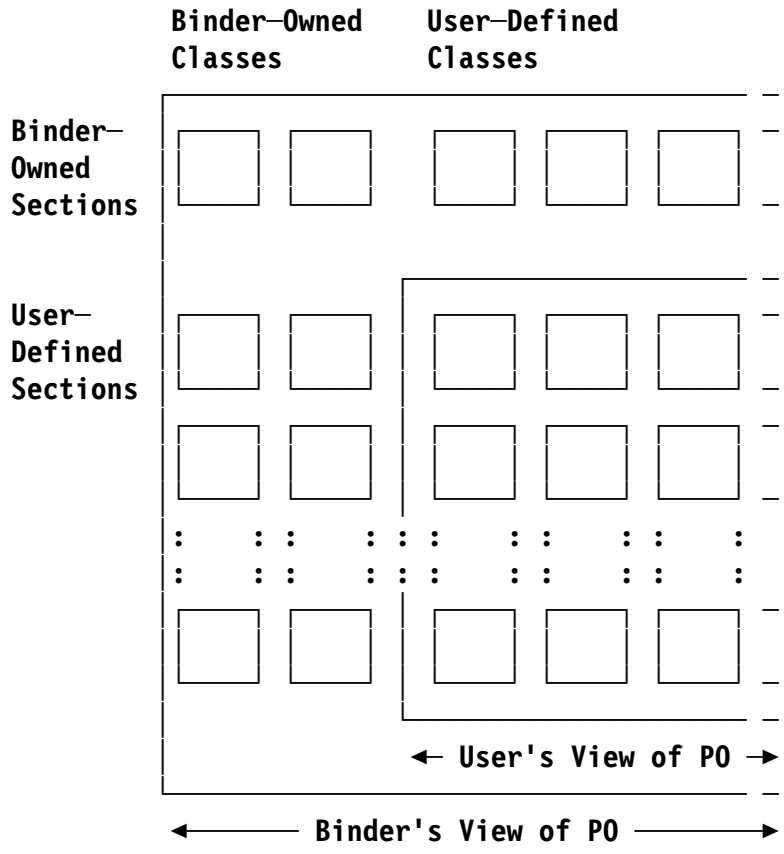
PO structure as seen by the translator and Binder user:

Section roughly equivalent to a "compilation unit"

- Consists of ***elements*** in various classes

MRG classes are constructed from ***Part Definitions*** and ***PseudoRegisters***

Binder Output view is more complex!



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

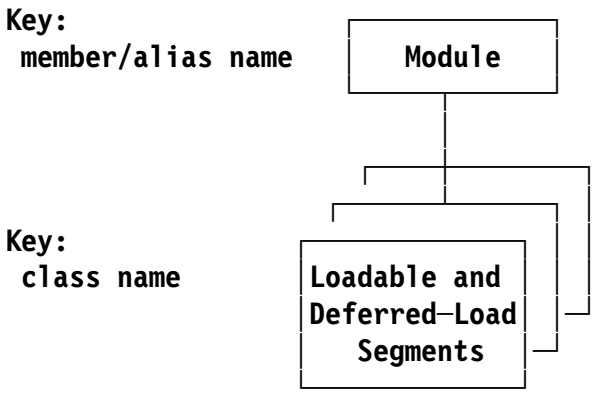
A segment may contain multiple classes if they have identical attributes

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 such as

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



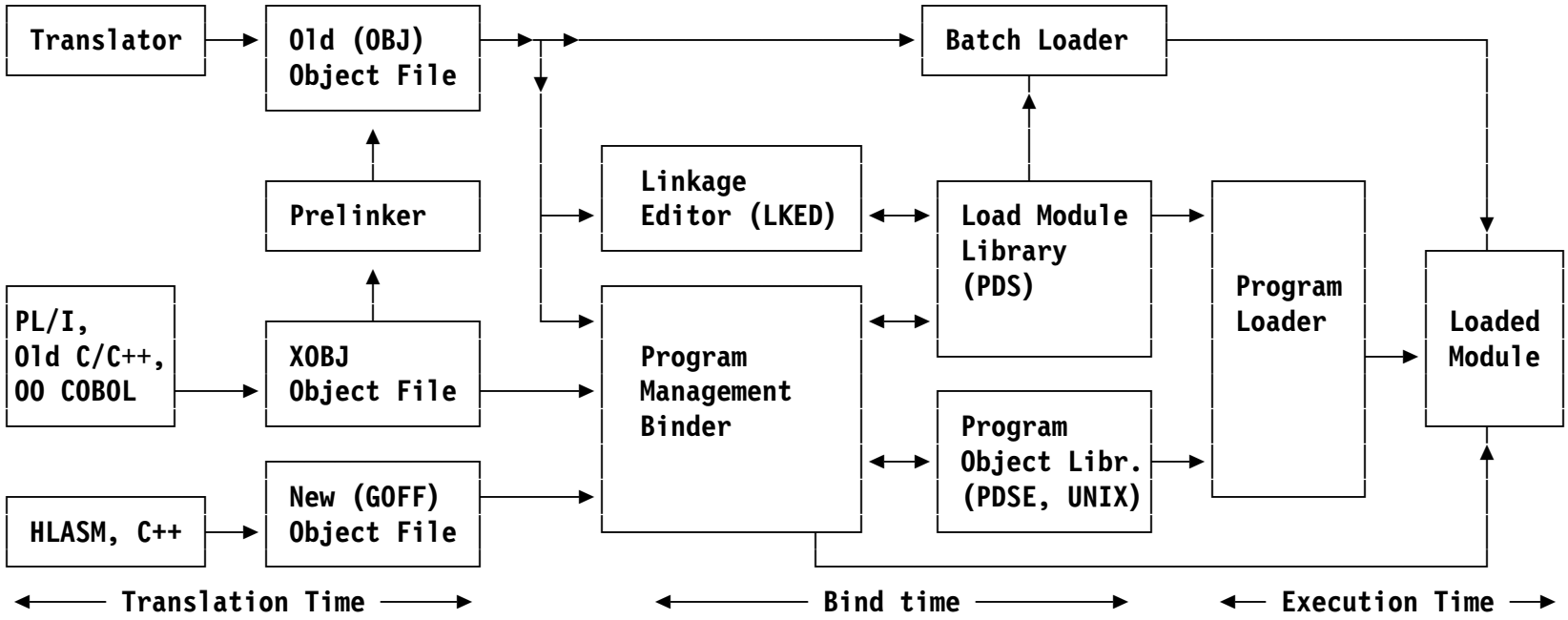
PO structure seen by PMLoader:

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



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 UNIX files

Can mix OBJ and GOFF to produce PO or LM (LM restricts features)

- "Source→OM→Binder→LM" equivalent to "Source→OM→LKED→LM"
- "Source→GOFF→Binder→LM" equivalent to "Source→OM→LKED→LM"

Can bind PO and LM to produce either (LM restricts features)

Dynamic Link Libraries (DLLs)

Dynamic linking: final resolution 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**

- Binder puts them in a “side file” for binding to other applications

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},dll_name,identifier
```

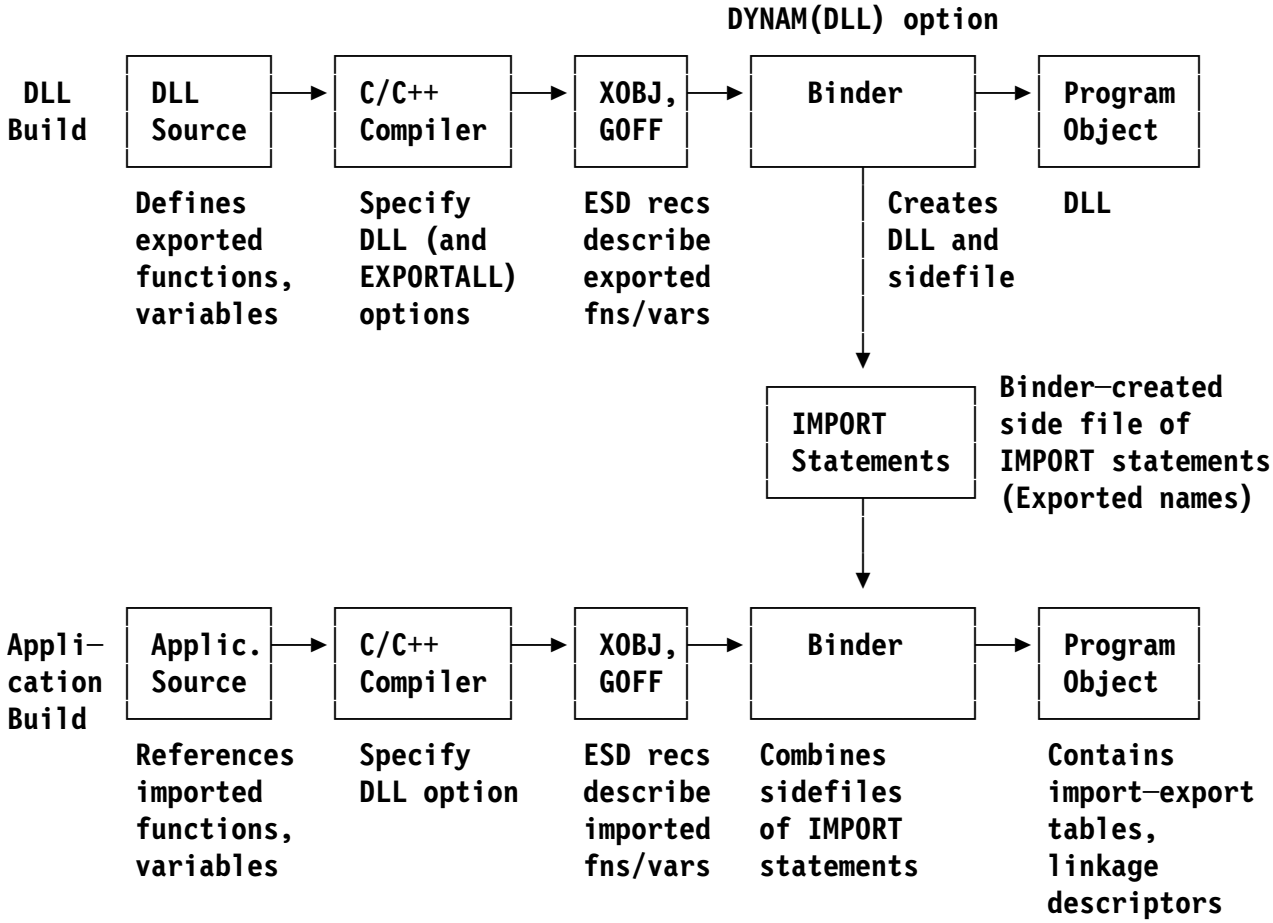
- Compilers (and HLASM XATTR statement) declare IMPORT/EXPORT status

Binder creates side file, import-export tables, linkage descriptors

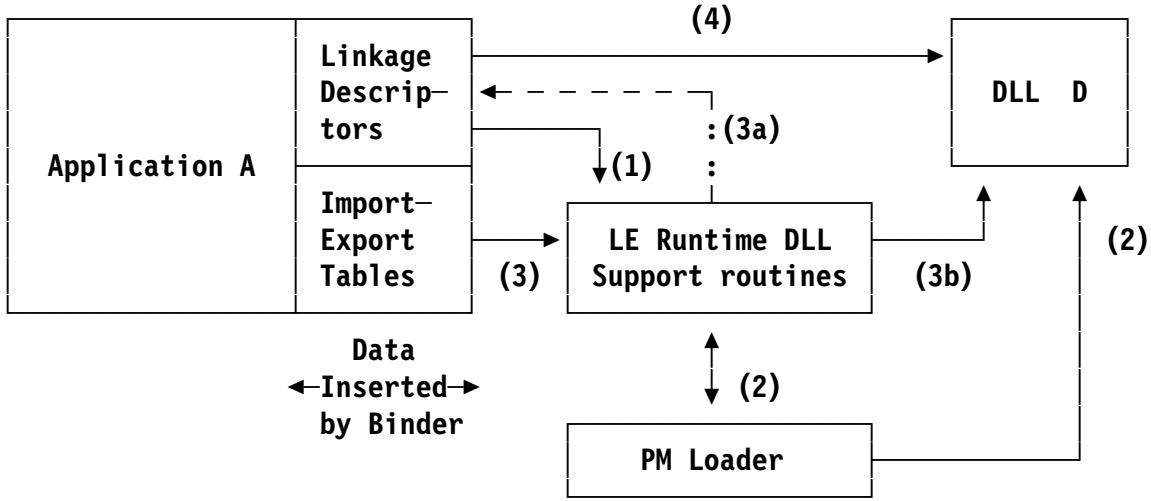
- DYNAM(DLL) option required for DLL creator and user

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 PMLoader to load the DLL

(3) LE uses import-export table to load DLL;

(a) then updates descriptors for code/data items to complete linkages

(4) Subsequent application references go directly to the requested (imported) name in the DLL

Summary

Binder and PMLoader support both load modules and program objects

	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	Prelinker not required
Extensibility	Not possible	Open-ended architecture

PDSE

- Can hold any record type
- No compression required; space reclaimed automatically
  - No gas (dead modules), no gaps (short blocks)
- Improved directory structure
  - Automatic expansion; not a fixed size
  - Indexed search (vs. sequential for PDS)
- No single-user ENQ for update
- Multiple simultaneous member updates
  - If same member, last STOW wins
- No sequential directory overwrite
- Long ALIAS names
- Holds only executable POs, or only other record types
- Utilize new hardware capabilities

Program Objects

- Split RMode: separate segments below/above 16MB “line” with inter-segment address resolution
- Faster loading via DIV mapping (except from z/OS UNIX files)
  - Several load-optimization options
  - No need to relocate the entire executable
- Functional superset of load module function
  - Two-dimensional class structure, determined by class attributes
  - Three class loading attributes
- Larger executables (1GB vs. 16MB)
- Long and mixed-case names (32K, vs. 8 upper-case)
- Auxiliary data preserved with the executable
- MODMAP option puts module map in section `IEWBMMP`, class `B_MODMAP`
- APIs for retrieving *all* data
- AMode specifiable on individual entry points

What is in object modules, and where they come from

How references are resolved to form executable programs

Structure of load modules and program objects, and how they are built

How modules are loaded into storage and relocated

How Dynamic Link Libraries are supported

Why using PDSEs and Program Objects is a good practice

For you: more flexibility in creating program structures

absolute loader
Places a module into storage at a predetermined fixed address, without relocating anything

ADATA Associated Data: information stored in a PO which is not required for binding, loading, or execution (e.g. debug data)

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 also *Part*)

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”

dynamic loading

Place modules into storage (with relocation, without linking) during program execution

dynamic linking

Place modules into storage ***with linking and relocation*** during program execution

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

entry table A special section inserted at the end of an overlay segment, to assist branching into other segments

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.

linking Resolve external names into offsets or addresses; combine multiple input name spaces into composite output name space

linking loader
Places modules into storage with linking, immediately prior to program execution

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 Parts in a deferred-load class are aligned before catenation with other Parts.

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

OBJ Traditional (since 1965) card-image object module format.

overlay A program structure allowing storage to be shared by different non-interacting parts of the program

overlay segment
The smallest separately loadable part of an overlay program, always loaded at the same offset from the module origin

Part Binding
In a MRG class, using CAT binding

Part, 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 and Commons, but not for PseudoRegisters.

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.

region a way to organize overlay programs into as many as four distinct overlay structures

relocate Assign actual-storage or module-origin-relative addresses to address constants

relocating loader
Places modules into storage ***and*** adjusts addresses to their actual “final” value

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

root segment
The lowest-addressed segment of an overlay program, always present during execution

scatter loading
The loading of module text into discontiguous 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 In a Program Object, 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. (See also *overlay segment*)

segment table
A special section describing segments of an overlay program, placed at the base of the root segment

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.

XOBJ OBJ format extended by C/C++ to handle long names and external symbol attributes. (Superseded by GOFF.)

1. z/OS MVS Program Management: User's Guide and Reference (SA22-7643)

2. z/OS MVS Program Management: Advanced Facilities (SA22-7644)

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, Program Logic manuals

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

5. High Level Assembler for z/OS, z/VM, and z/VSE Language Reference (SC26-4940)

6. High Level Assembler for z/OS, z/VM, and z/VSE Programmer's Guide (SC26-4941)