How the Linkage Editor Works:
A Tutorial on Object/Load Modules,
Link Editors, Loaders, and
What They Do for (and to) You

SHARE 96 (Feb. 2001), Session 8169

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

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

March 1, 2001

Table of Contents

Table of Contents

Table of Contents

Introduction

Note: This is not a tutorial on Link Editor usage!

- What happens to programs "on the way to execution"

- Why program linking is needed

- What assemblers and compilers produce: object modules

- What program linking does with object modules

- Saving the results of linking: load modules

- What happens when load modules are put into storage

- Why the Linkage Editor and Loader are the way they are

- The future: the good things the new Binder does for you

- Glossary and references

1. The Beginner's View

```
                    ┌─────────────────────────────┐
┌──────────────┐    │        The COMPUTER         │    ┌──────────────┐
│              │    │                             │    │              │
│  My Program  │───▶│  Something Magical Happens  │───▶│  My Output   │
│              │    │                             │    │              │
└──────────────┘    └─────────────────────────────┘    └──────────────┘
```
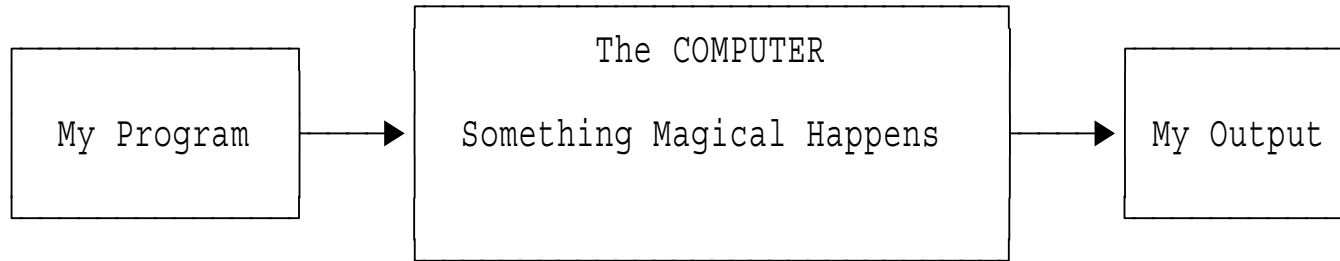
2. The After-a-Little-Experience View

```
          ┌────────────────────────────────────────────────────┐
          │                   The COMPUTER                      │
          │  ┌──────────────┐    ┌────────────────────┐         │
┌─────────┐  │              │    │ Something Very     │         │  ┌──────────┐
│A Source │  │  Language    │    │ Peculiar and       │         │  │          │
│Program  │─▶│  Compiler    │───▶│ Mysterious         │─────────┼─▶│ My Output│
│         │  │              │    │ Happens Here       │         │  │          │
└─────────┘  └──────────────┘    └────────────────────┘         │  └──────────┘
          └────────────────────────────────────────────────────┘
```
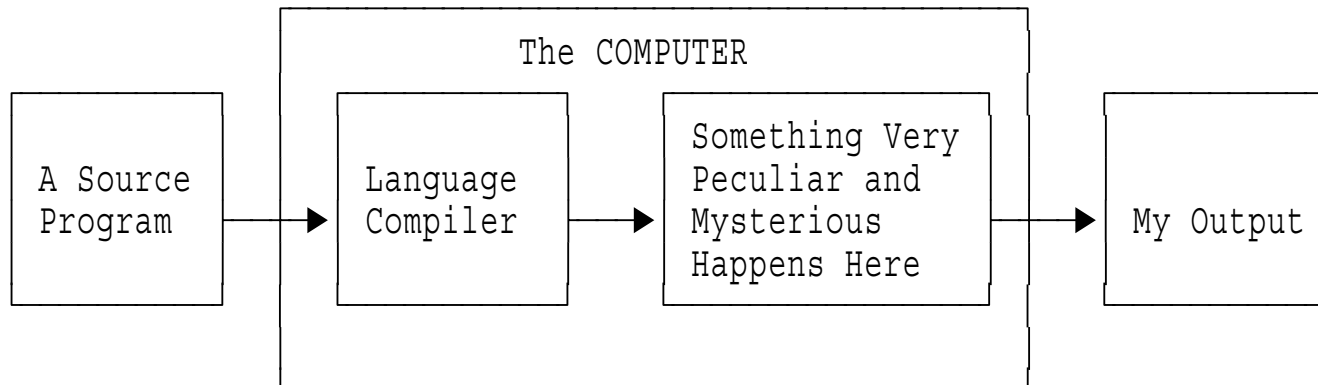
- We learn to distinguish between compile time and run time

3. The After-Some-More-Experience View

```
+-------------+     +-------------+  +-------------+  +-------------+     +-------------+
| Source      |     | Language    |  | Linker      |  | Program     |     | Output      |
| Program     | --> | Compiler    |->|             |->| Execu-      | --> |             |
|             |     |             |  |             |  | tion        |     |             |
+-------------+     +-------------+  +-------------+  +-------------+     +-------------+
                       Compile          Link             Run
                        Time            Time             Time
```
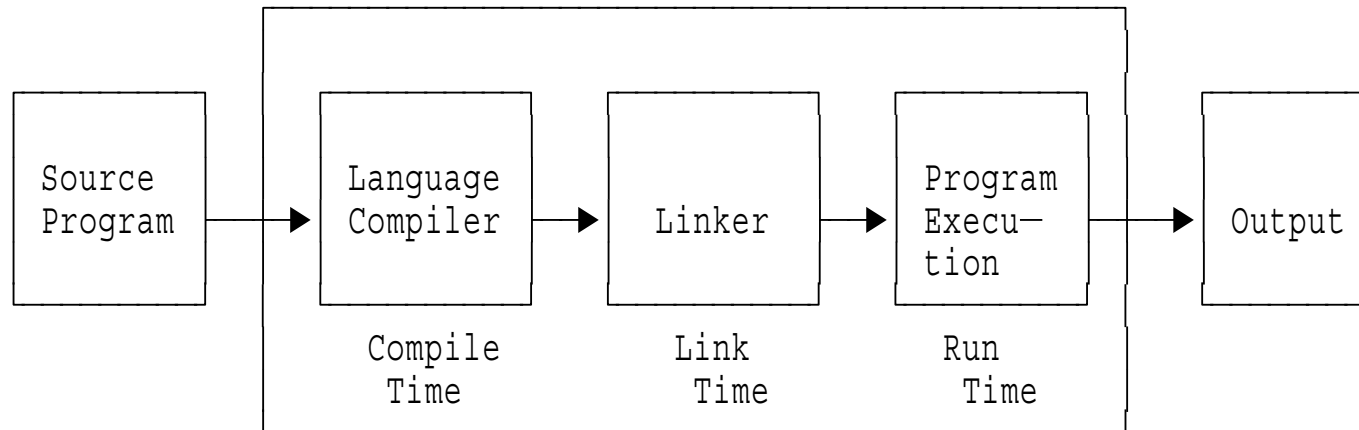
- We learn to distinguish among compile, link, and run times

4. Our View

```
                    ┌──────────────────────────────────────────┐
                    │      ┌──────────┐    ┌──────────┐    ┌──────────┐              │
                    │      │ Linkage  │───▶│  Load    │───▶│ Program  │──────┐       │
 ┌──────────┐       │      │ Editor   │    │ Module   │    │ Fetch    │      │    ┌──────────┐
 │ Source   │       │  ┌────────┐  └──────────┘    └──────────┘    └──────────┘      │    │ Program  │
 │ Trans-   │───────┼─▶│ Object │──┘                                             └───▶│ Execution│
 │ lator    │       │  │ Module │──┐                 ┌──────────────┐                 │ and Output│
 └──────────┘       │  └────────┘  └────────────────▶│ Batch Loader │─────────────────┘    └──────────┘
                    │                                └──────────────┘              │
                    └──────────────────────────────────────────┘

  Compile                        Link Time                              Run Time
   Time
```
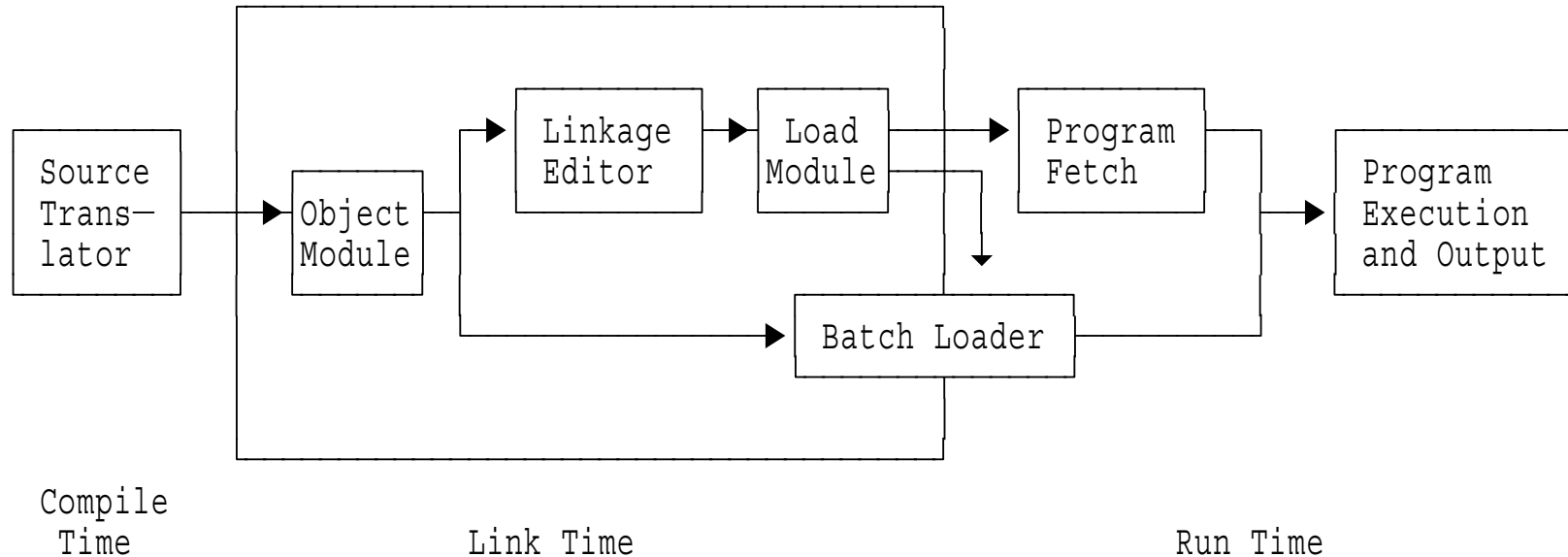
- Our focus will be almost entirely on the items in the "Link Time" box, plus Program Fetch

- We will refer to some compile-time and run-time topics and issues

- Anything that gets "big" or "complicated" is hard to manage

- The world's oldest paradigm for handling big, complex problems:

  - "Divide and Conquer": break the problem into manageable pieces

  - Many dignified names have been given to this: Analysis, Modular Decomposition, Top-Down Analysis, Program Partitioning, Structured Programming...

  - As your mother told you,
    "Don't try to eat that whole thing! Cut it into pieces first!"

- Naturally leads to the question:

  - How do I put the divided and conquered pieces back together again?

  - "Synthesis" is the dignified name

  - As your mother told you,
    "If you took it apart, it's up to you to put it back together!"

- Program linking and loading are fundamental to any system

  - Linker capabilities (or shortcomings) have profound and widespread impacts

- Putting the pieces back together ("binding") can occur at many times

  - Compile time: compile all needed items from source (early Algol, Pascal)

  - Link Edit (pre-execution) time: everything "bound" prior to execution ("static binding")

  - Program initiation time: everything "bound" at start of execution

  - Execution time: pieces "bound" only if required

- Choice of "binding time" implies trade-offs:

  - Earlier times: efficiency vs. inflexibility

  - Later times: efficiency, flexibility, modifiability vs. costs

  - "Efficiency" is measured in many dimensions...!

- Program re-composition requires additional information:

  - A way to name the pieces to be bound

  - A way for the pieces to refer to one another

© IBM Corporation 1994, 2001

- In this discussion:

  - Information to assist with "re-composition" (or "binding")

    — **External names**: used to name the pieces to be bound

    — **External names, address constants**: how the pieces refer to one another

    — N.B.: The two types of external names need not be the same!

  - External names are the critical element in linking!

- Our concerns, and the program re-composition tools involved:

  - Link-edit (pre-execution) time: Linkage Editor

  - Program initiation time: Batch Loader

  - Execution time: Operating System Program Fetch services

- Understanding the pieces, and how they were bound

  - Link Editor and Batch Loader MAPs? AMBLIST?

  - Binder is much more informative (more about this, later)

Translator Output: Object Modules

- For the truly exciting details, see the References (slide 67):

  – High Level Assembler for MVS & VM & VSE *Programmer's Guide*

  – OS/390 DFSMS *Program Management*

- Control Section (or "Section" or **CSECT**, for short)

  – A set of program elements bearing *fixed* positional relationships to one another

  – A unit whose addressing and/or placement relative to all other Control Sections does not affect the program's run-time logic

  – The basic unit of program linking

  – Types: Ordinary (**CSECT**), Read-Only (**RSECT**), Common (**COM**)

- External Symbol (a "Public" symbol; internal symbols are "Private")

  – A name known at program linking time

  – A symbol whose value is intentionally not resolved at translation time

- Address Constant ("Adcon")

  – A field within a Control Section into which an actual address (most of the time!) will be placed during program relocation

- Pseudo-Register (or, External Dummy Section)

  – PR names may be identical to other External Symbol names without conflict

  – More about these, at slides 34 and 37

- 80-character (card-image) records, with 3-character ID in columns 2-4

  ESD   External Symbol Dictionary (symbols and their attributes)

  TXT   Machine Language instructions and data ("Text")

  RLD   Relocation Dictionary (data about adddress constants)

  SYM   Internal ("Private") Symbols

  END   End of Object Module, with **IDR** (Identification Record) data

- One object module per Compilation Unit

- "Batch" translations may produce multiple object modules

- Describes **external symbols** (1 to 3 16-byte items per record)
- Numbered sequentially within each object module, starting at 1
  - The (16-bit) number is called the **ESDID**
- Four basic types of external symbol:

SD,CM Section Definition: the name of a control section
- Data: name, ESDID, length, section-origin address, AMODE & RMODE
- Blank **SD** name sometimes called Private Code (**PC**)
- Common (**CM**), **SD** items handled differently (CM has length, <u>no</u> TXT)

LD Label Definition: the name of a position at a fixed offset within a Control Section; typically, an Entry Point
- Data: name, address of the label, and ESDID of the section it's in (LDs don't have a separate ESDID of their own)

ER,WX External Reference: the name of a symbol defined "elsewhere" to which this module wants to refer
- Data: name, ESDID

PR Pseudo Register: name of a Pseudo Register (the Assembler calls it **XD**, External Dummy Section)
- Data: name, ESDID, PR length and alignment requirement

- ESD records must appear first in each object module

- ESD items originate in various language constructs, such as:

ESD item	Assembler	VS Fortran	OS PL/I	COBOL	C/370
SD	Csect, Rsect	Routine, Block Data	Procedure External vars, if INIT	Outermost program	R/W data
CM	Com	Common	External static if no INIT		
ER	Extrn, V-con	Call, Common	Call, data reference	Static Call Literal unless DYNAM option	Call, data reference
LD	Entry	Entry	Entry	Entry	Function
PR, XD	DXD, Q-con + Dsect		File, Fetchable, Controlled		Writable static for RENT (not in final LM)
WX	Wxtrn		Used, but not in the language		

- Sample Assembler Language program:

```
                    Source Program

        ┌─────────────────────────┐
        │ AA    CSect              │          Control Section
        │ *                        │ ┐      External symbols
        │       Extrn X            │ │          External reference
        │       Entry E            │ ┘          Entry—Point symbol
        │ *                        │
        │ E     DC     A(X+44)     │ ┐        Address constant
Internal ┌│ Con1  DC     F'1'        │ ┤ Text
Symbols └│ QCon  DC     Q(D)        │ ┘        Address constant
External ─│ D     DXD    A           │          External Dummy (Pseudo—Register)
Symbol    │       End                │
        └─────────────────────────┘
```

- Assembler's External Symbol Dictionary (ESD) listing

```
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
```

  - Note: D's "address" field is an alignment mask (3 = fullword)

- Contains **machine language** instructions and data

  - Up to 56 bytes per record

- Data:

  1. How many bytes of text data are in this record

  2. ESDID of the control section it belongs in

  3. Address within that control section where the text is to be placed

- Always a contiguous string of bytes

  - Discontinuities in the "text" stream start a new TXT record

  - Contents of gaps not specified

    — Early days: storage not zeroed, so gaps might contain oddities

      - Some codes (regrettably) depend on zero initial storage

- Packed stream of 4-byte or 8-byte RLD items

- Information about relocatable (and Q, CXD) **address constants**
  - Where the constant is to be found
  - What value should be in the constant (what it should point to)

- Each RLD item has 6 pieces of information:

  1. **R Pointer**: ESDID of the name whose "target address" the adcon will contain
     - I.e., what it points to; where to get the "relocation value"

  2. **P Pointer**: ESDID of the section where the constant resides
     - I.e., where to find the constant itself (but not what will be in it)

  3. **Address**: the address or offset at which the constant resides within its section (as specified by the P pointer); TXT may contain data to be added to the relocation value

  4. **Length**: the constant's length (in bytes)

  5. **Type**: whether it's an A-type (data), V-type (branch, possibly *indirect*), Q-type (PR offset), or CXD (PR "Cumulative Length") (More on Q/CXD later...)

     Warning!! A- and V-type constants can behave *very* differently! (More later...)

  6. **Direction**: for A-type adcons, whether to add or subtract the relocation value

- Last three items are encoded in a single byte

- Assembler program (same as on slide 14):

```
              Source Program

      ┌─────────────────────────────┐
      │ AA    CSect        │              Control Section
      │ *                  ├── External symbols
      │       Extrn X      │                  External reference
      │       Entry E      ┘                  Entry–Point symbol
      │ *                  │
      │ E     DC    A(X+44)│┐             Address constant
Internal ┌│ Con1  DC    F'1'   │├ Text
Symbols  └│ QCon  DC    Q(D)   │┘             Address constant
External ─ │ D     DXD   A      │              External Dummy (Pseudo–Register)
  Symbol   │       End          │
      └─────────────────────────────┘
```
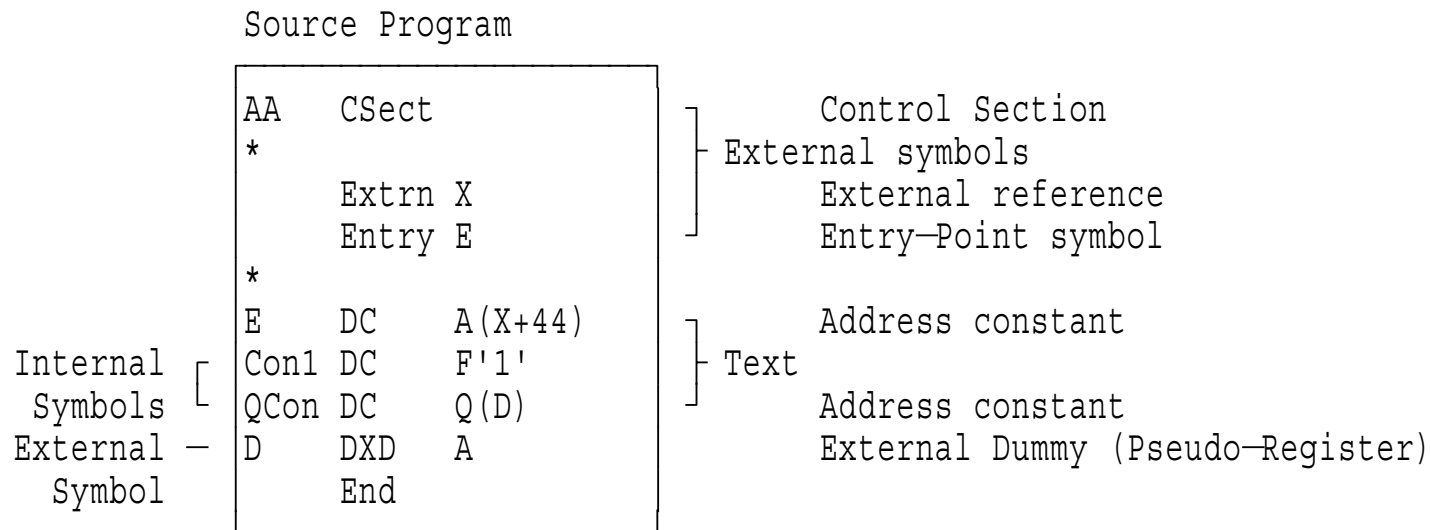
- Assembler's Relocation Dictionary listing (see ESD on slide 14)

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

- Contains **internal symbols** used by source translator

  – Produced by Assembler, VS Fortran

- SYM information is (sometimes) useful for debugging

  – TSO TEST, some commercial debuggers

- Ghastly bit-squeezing packed format (details are truly impressive)

  – Maximum symbol length is 8 characters

  – See HLASM Programmer's Guide (SC26-4941), Appendix C

- Linkage Editor doesn't make SYM records convenient to use

  – Copies SYM (and SD,CM info from ESD) records to front of load module

  – No system facilities for retrieving them easily!

- Recommend using High Level Assembler SYSADATA output instead

  – More information, in a more usable and accessible format

- Required; primary function is to signal the end of the object module

- Some additional (optional) information may be provided:

  1. Requested ("nominated") execution-time entry point

     - By ESDID and address, or by external name

     - These requests may be overridden by other factors or controls

  2. Actual length of a Control Section whose length was not specified on its ESD record

     - This feature saves effort in some compilers

  3. Identification (IDR) data (0, 1, or 2 19-byte IDR items)

     - Translator's product number, with version and modification level

     - Date (YYDDD format) of the translation

     - Use windowing for dates past year 2000 (**YY < 65** means **20YY**)

- CMS LOAD has meager control-statement capabilities

  – Only ENTRY and LIBRARY statements

- Object-like records can be used for some control functions

  REP   Replacement text: behaves like a TXT record, but hex values are specified in EBCDIC for ease of preparation

        – Also used by the VSE Linkage Editor

  LDT   Loader Terminate: last record of a group of object modules, with optional indication of an entry address and SETSSI info

  ICS   Include Control Section: placed ahead of an object module to override the original length of a named control section

  SLC   Set Location Counter: sets the (absolute virtual) load address at which the following modules will start loading

  SPB   Set Page Boundary: sets the loader's location counter to the next page boundary; may appear before/after any module

- See the CMS LOAD command description for further details

Combining Object Modules with the Batch Loader

- A simple example of initiation-time linking

- Illustrates the basic principles involved in linking

  - Applicable to CMS, also

- The Batch Loader can do a lot more than this example shows

  - A one-pass processor, with address "fixups"

- Suppose a program consists of two source modules:

```
        Module 1                          Module 2
   Loc                                Loc
   000 │MAIN                          000 │SUB
   │            COMMON /WORK/ ...      │            COMMON /WORK/ ...
   │            — — —                  │            EXTERNAL ZDATA
   │            CALL SUB               │            — — —
   │            — — —                  700 │        Addr(WORK)
   200 │        Addr(SUB)              704 │        Addr(ZDATA)
   204 │        Addr(WORK)             │            — — —
   208 │        Addr(ZDATA)
   │            — — —
   │            ENTRY ZDATA
   260 │ZDATA                          (For this example, values
   │            — — —                   are given in decimal)
```

  - Program MAIN contains a ZDATA entry point, and refers to the COMMON
    area named WORK

  - Subprogram SUB refers to the external name ZDATA, 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 ZDATA Addr=260                  LD for Entry ZDATA, ESDID=1, Addr=260
ESD ER ID=3 SUB                             ER for reference to SUB, ESDID=3
TXT    ID=1 Addr=000      'abcdefghijk'     Text for MAIN, address 000
TXT    ID=1 ...            etc.             Text for MAIN
TXT    ID=1 Addr=100      'mnopqrstuvw'     Text for MAIN, address 100
TXT    ID=1 Addr=208       00000260         Text has offset in MAIN for internal adcon
TXT    ID=1 Addr=260      '01234567890'     Text for MAIN, address 260
TXT    ID=1 ...            etc.             Text for 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(ZDATA)
END    Entry=MAIN                           End of module, nominated entryname=MAIN
```

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

- RLD contains information about three address constants

  − TXT for Addr(ZDATA) 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 ZDATA                                     ER for reference to ZDATA, ESDID=3
TXT    ID=1 Addr=040        'qweruiopasd'             Text for SUB, address 040
TXT    ID=1 ...             etc.                      Text for SUB
TXT    ID=1 Addr=180        'jklzxcvbnm'              Text for SUB, address 180
TXT    ID=1 ...             etc.                      Text for 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(ZDATA)
END                                                   End of module
```

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

- RLD contains information about two address constants

- 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**/ **subtracting** relocation value to A-con P-field contents; **storing** it in V-cons

  5. Sets entry point address and enters loaded program

- The linked program is not saved

- Composite ESD

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

(For this example, values are given in decimal)

- The resulting program, loaded into storage for execution:

```
123500 (MAIN)                123800 (SUB)            124600 (WORK)   125200
                                                                     (end)

 abcdefghijk...              ...text...              length=600
 ...more text...             qweruiopasd...
 ...more text...             ...text...
 mnopqrstuvw...              ...text...
 ...                         jklzxcvbnm...
                             ...text...
       A(SUB) 00123800
      A(WORK) 00124600
     A(ZDATA) 00123760

          Adcons               00124600  A(WORK)
                               00123760  A(ZDATA)
 (ZDATA)
 01234567890...                 Adcons
 ...                         ...
```
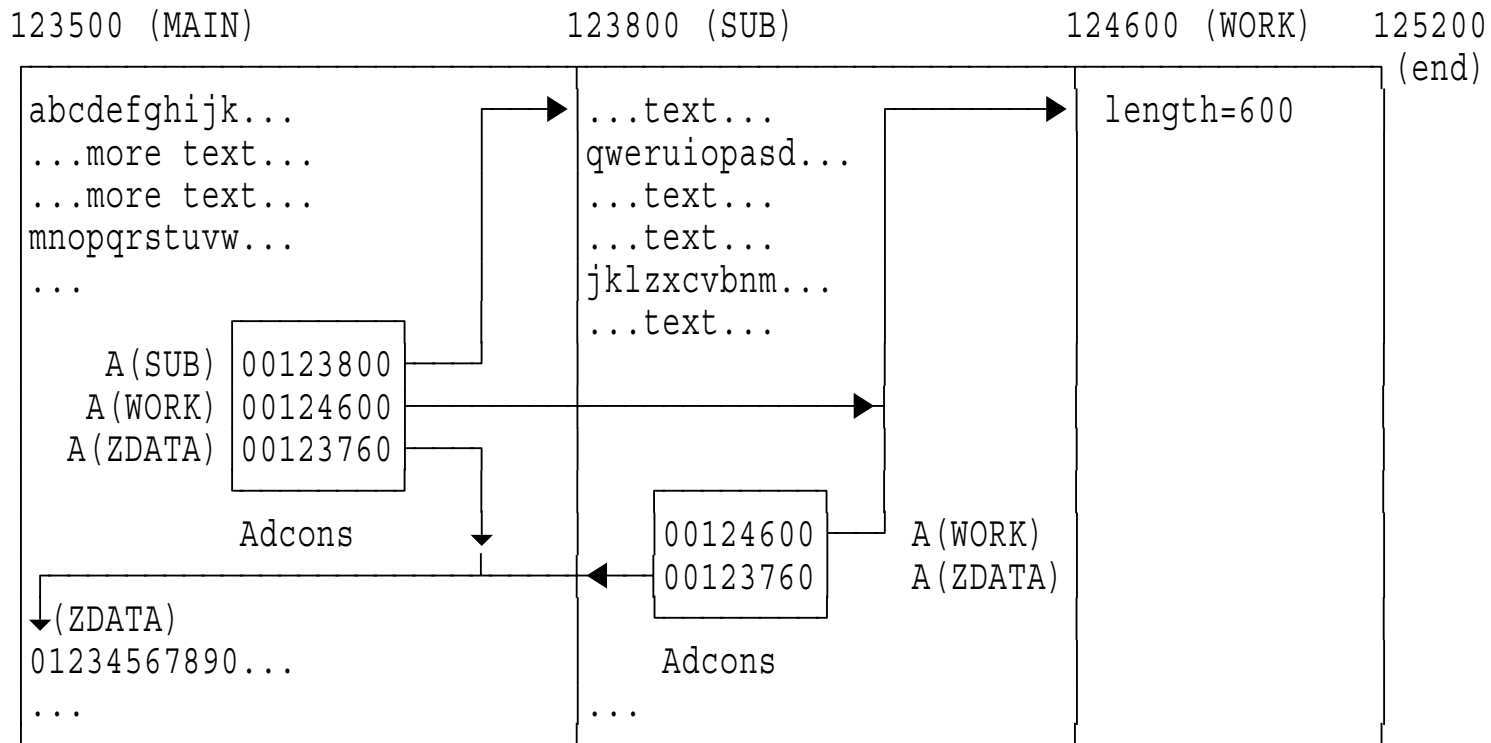
- Storage was allocated for three control sections (two SD, one CM)

- 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 "load address" for load modules is zero

  - Program (CESD, text, RLD, other info) written to DASD

  - Unresolved ERs OK if NCAL specified

  - Final relocation will be done by Program Fetch (slide 46)

- Basic executable unit for MVS-like systems

    – The world's longest-surviving form of "executable binary!"

- Designed (a long time ago) for

    1. Loading into storage with minimal overhead

        – Binary (zero-origin) program image, requiring only relocation

    2. Editing

        – Retains enough information to permit

            — Replacement of any component

            — Restructuring of the entire module

            — Renaming of (almost!) any element

        – Unless you tell the Linkage Editor not to keep it! (NE option)

    3. Minimal run-time storage requirements

        – Only "necessary" items are in storage

        – Complex overlay structures are supported

- Load module's structure very similar to object module's
    - Simplifies processing of each
- Basic contents (analogous to object module records)

CESD	Composite External Symbol Dictionary
Text	Machine language instructions and data
RLD	Relocation Dictionary
SYM	Object-module records copied directly into load modules
IDR	Identification records (from object modules, Linkage Editor, user, and ZAP)
EOM	End of module (a flag field on a CTL 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 is called "record format," "block format" or "block loaded"

SYM	Only if TEST option, and SYM info is present in inputs (May also contain some ESD data)
CESD	Omitted if NE option
IDR	
CTL	Info about first text record
Text	First text record, placed at module load address
CTL/RLD	Relocate first text record, and read second text record at specified offset from module load address
: :	
Text	Last text record
EOM/RLD	Relocate last text record, signal end of module

* →

```
* Location of first text record kept in PDS directory
  (essentially, a copy of info from the first control record)
```

- Inputs

  - Object modules

  - Load modules

  - Control statements to direct the Linkage Editor

    — Where to get additional inputs:
    INCLUDE, LIBRARY

    — What to do with all the pieces:
    REPLACE, CHANGE, INSERT, ORDER, PAGE, OVERLAY, EXPAND

    — How to describe and name the output module:
    ENTRY, NAME, SETSSI, IDENTIFY, SETCODE, MODE, ALIAS

- Outputs

  - Load module(s)

  - Listing, terminal messages

- Processing: two passes over inputs (described on slide 32)

  - Very much like an assembler!

- Pass 1

  – Read all inputs (explicitly or implicitly designated)

    — If not NCAL, unresolved ERs cause library search
    — WX never causes library search; resolved only if symbol is already present

  – Build symbol table (CESD) by merging ESD/CESD items from all inputs

  – Determine lengths, orderings, offsets, etc.

    — First SD wins, longest CM wins, all nonzero-length PC items kept, etc.
    — CMs matching an SD become that SD

- Intermediate processing

  – Resolve interdependences (CMs mapped at end to save DASD space)

  – Assign addresses relative to zero module origin, relocate adcons

    — CMs assigned at end; don't have to be written out (no text)

  – Assign module's entry point (complex rules)

  – Write module MAP (and XREF, if entire module is in storage)

- Pass 2

  – Write out all the pieces in the correct order, with relocation data

    — If NE (not editable) option: no CESD, IDR, or SYM

  – STOW directory entry (or entries, if ALIASes)

  – Write XREF (if module didn't fit in storage)

- 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: Link Editor may write short text-record blocks
  - Large gaps: Link Editor writes out the preceding text record
  - Only one partial CSECT allowed per block
  - Also depends on space left on track (impenetrable algorithm decides)

- Initial values: what eventually appears in the gaps?
  - Small gaps: depends on what is in the Link Editor's text buffers
    — For early releases, it could be anything (buffers weren't cleared)
      - Now cleared to zeros
    — Binder's FILL option lets you specify a value
      - Can help debug unitinialized-variable problems

  - Large gaps: depends on what's in storage during module Fetch
- Advice: <u>never</u> depend on anything <u>you</u> didn't initialize

- Allow sharing by name of dynamically managed external objects defined in separately translated <u>re-entrant</u> programs

  – Originally required by 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"

  – Hence the Assembler's name, "External Dummy" (XD)

  – The dummy section is also called the "Pseudo-Register Vector" (PL/I's PRV allowed up to 1024 more 32-bit "registers")

    — A template; a data-structure mapping created at link time

  – Example on slide 36

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

  - If multiple definitions, longest length and strictest alignment win

  - Accumulated length/alignment of PRV items then determine offsets 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 <u>link time</u>

- Runtime code must acquire a storage area of the CXD 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 addr(File 1's Control Block)
```

- Link with other modules; Link Editor creates "virtual" PRV

```
┌ ─ ─ ─ ─ ─ ┐  ┐ ┐
:           :  │ │
├ ─ ─ ─ ─ ─ ┤  │ │ ↓  ← Offset of FILE1CB's PR is placed in Q(FILE1CB)
│           │  │ │
│ FILE1CB's PR │  │ │
├ ─ ─ ─ ─ ─ ┤  │ │
:           : :│ :
│           │ :│ ↓  ← PRV Length is placed in CXD
└ ─ ─ ─ ─ ─ ┘ ↓
```
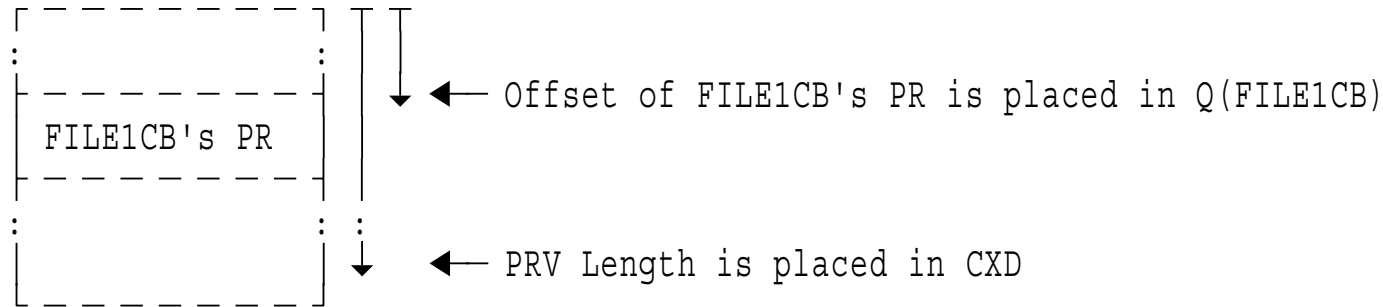
- Main program acquires storage for <u>real</u> PRV

```
        L    0,PRVLen    Get length of PRV
        GetMain R,LV=(0) Get storage
        LR   11,1        Carry PRV address in R11
        ─ ─ ─
    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 Pseudo-Registers have similarities and differences

	COMMONs	**Pseudo-Registers**
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 (unless an SD)	Run-time code's responsibility
Copies	One per load module	One PRV per module instantiation
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

- 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 $\leq$ 244; CESD $\leq$ 248; IDR, CTL, RLD $\leq$ 256; Text $\leq$ track length
    (Can force text records to be <u>much</u> shorter than a track)

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

- "Directory name space" (PDS directory names) unrelated to external (CESD) names (which may be unrelated to internal names, too!)

  - Can assign member and alias names unrelated to CESD names

    — Object module item named AA, renamed to BB in load module, PDS member is CC

  - TSS Linkage Editor didn't allow this confusion

Overlay Modules

- Overlays are more complex than block-format modules

    - Different parts of a module may share the same storage (at different times!)

    - Require special Linkage Editor considerations

- 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
```
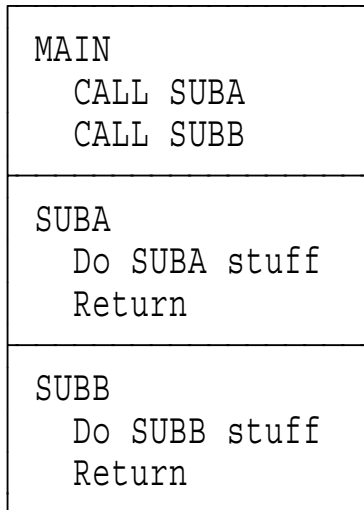
- SUBA and SUBB might be overlaid, like this:

```
MAIN
   CALL SUBA
   CALL SUBB
```
```
SUBA                    SUBB
 Do SUBA stuff           Do SUBB stuff
 Return                  Return
```
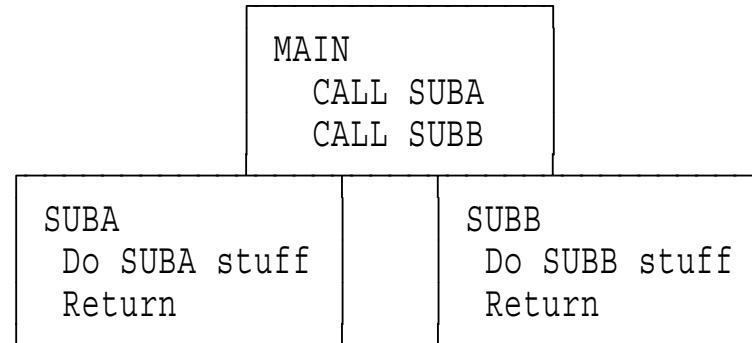
- SUBA and SUBB share the same storage

- 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

- Root segment is always present

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

```
                          ┌──────────┐              ─┐
                          │ SEGTAB   │               │
                          ├──────────┤            Root
         Entry ───►       │ MAIN     │            Seg-
         Point            ├ ─ ─ ─ ─ ─┤            ment
                          │ AUTO1    │               │
                          │ AUTO2    │               │
                   ┌──────┴──────────┴──────┐       ─┘
                   │◄──── Overlay A ────►│
              ┌────┴─────┐          ┌─────┴────┐
              │ SUBA     │          │ SUBB     │
              ├──────────┤          └─────┬────┘
              │ SUBJ     │        ┌───────┴──────────┐
              └────┬─────┘        │◄── Overlay C ──►│
           ┌───────┴──────────┐   ├──────────┬───────┤
           │◄── Overlay B ──►│   │ SUBV     │ SUBW  │
     ┌─────┴────┬────────────┐   │          │       │
     │ SUBR     │  SUBS      │   └──────────┘       │
     └──────────┴─────┬──────┘
                      │ SUBT │
                      └──────┘
```

Root Segment

```
SEGTAB

MAIN
V(SUBA)      (1)
V(SUBB)

ENTAB1       Entry Table (ENTAB)
A(SUBA)
A(SUBB)          (3)
SVC
```

```
SUBA                    SUBB
V(SUBS)   (1)
V(SUBR)
V(SUBJ)   (2)           V(SUBV)   (1)
                        V(SUBW)
SUBJ
                        ENTAB3    Entry
ENTAB2    Entry         A(SUBV)   Table
A(SUBT)   Table         A(SUBW)     (3)
A(SUBR)                 SVC
A(SUBS)
SVC                     SUBV      SUBW

SUBS      SUBR          A(WORK)
V(SUBT)                 WORK      (4)
          V(SUBJ)  (2)
SUBT  (2)
```

- 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, <u>not</u> 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 <u>always</u> 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!

- Overlays can be arranged in independent groups: REGIONs

  – Allows greater freedom in structuring programs

- Each region can be an overlay structure!

  – Four regions allowed

- A form of dynamic loading

  – Specific routines loaded as needed

  – No displacement of other segments

- Example with two regions:

```
─────────── Region 1 ───────────

                  ┌──────────┐          ┐
                  │  SEGTAB  │          │
                  ├──────────┤          │ Root
        Entry ───▶│   MAIN   │          │ Seg-
        Point     ├ ─ ─ ─ ─ ─┤          │ ment
                  │  AUTO1   │          │
                  │  AUTO2   │          │
              ┌───┴──────────┴───┐      │
              │◀─── Overlay A ───▶│      │
              ├──────────┬────────┤      ┘
              │  SUBA    │  SUBB  │
              ├──────────┤  ──────┴──────┐
              │  SUBJ    │ │◀─ Overlay C ─▶│
          ┌───┴──────────┴─┐├───────┬───────┤
          │◀─ Overlay B ──▶││ SUBV  │ SUBW  │
          ├───────┬────────┤└───────┴───────┘
          │ SUBR  │  SUBS  │
          └───────┤ ───────┴─┐
                  │  SUBT    │
                  └──────────┘

─────────── Region 2 ───────────

              ┌───────────────────────┐
              │◀── Overlay W(REGION) ──▶│
              ├──────────┬────────────┤
              │  SUB1    │   SUB3     │
              ├──────────┤  ──────────┴──┐
              │  SUB2    │ │◀─ Overlay Y ─▶│
          ┌───┴──────────┴─┐├───────┬──────┤
          │◀─ Overlay X ──▶││ SUB6  │ SUB7 │
          ├───────┬────────┤└───────┴──────┘
          │ SUB4  │  SUB5  │
          └───────┴────────┘
```
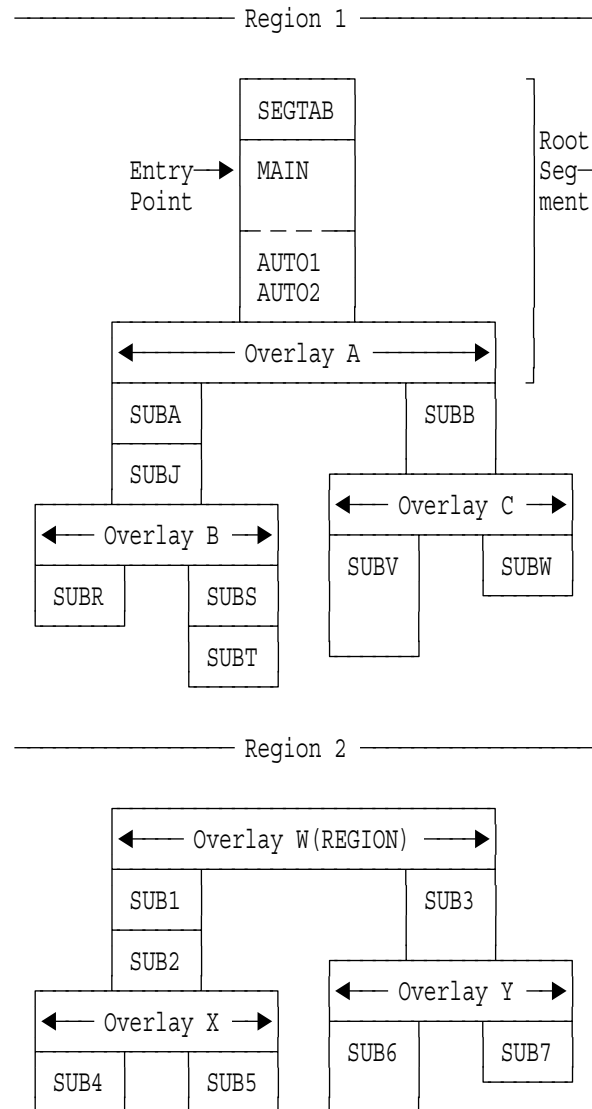
- Pro:

  - 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!

- 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 <u>indirect</u>! (A-type is always "direct")

  - Additional overhead in calls to segments needing to be loaded

  - Calls among certain modules may be forbidden (or wrong)

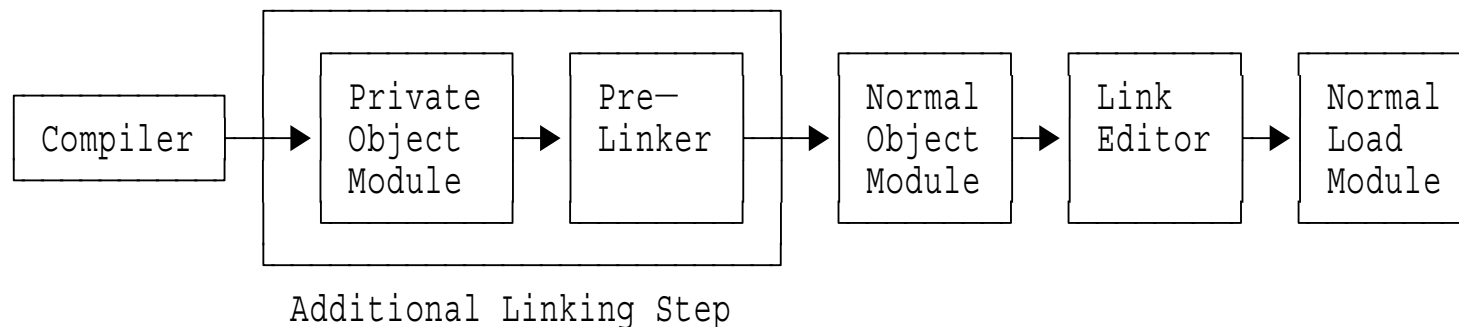# Bringing Load Modules into Storage: Program Fetch

- Used for all module loading from disk (LOAD, LINK, XCTL, ...)

  - Except during IPL...

- Skip over everything preceding the first control record

  - SYM, IDR, CESD (PDS directory info makes the skipping simple)
  - Therefore, no linking! (CESD info has been ignored)

- Control records tell length and relative address of following text record

  - 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

  - R and P pointers ignored
  - Q-cons and CXD were completed at linkage-edit time

- Note: two stages of relocation are involved:

  1. Linkage Editor relocates addresses relative to zero module origin

  2. Program Fetch relocates addresses relative to module's "load address"

- Overlay Supervisor

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

Looking Backward

- Linkage Editor

    – Written in 1964-65 by small team in IBM Poughkeepsie

        — Coding standards and techniques were still being developed...

    – Program Fetch, Overlay Supervisor done at the same time

        — PDS's, BLDL, STOW, etc. added to OS in response to LKED needs

        — Initial release ran in 18KB (core was expensive; 32KB machines were *big!*)

- OS Batch Loader

    – Appeared much later (about 1969) with OS/360 Release 17

- Very advanced technology for that time

    – Very rich functionality

    – No built-in programming model (like past systems)

- 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 within the module to make editing possible
- DASD is slow, and central storage is precious and expensive
  - Therefore: short records are a good thing
  - Therefore: packing module pieces tightly is a good thing
  - Therefore: overlay structures are a very good thing
- 24-bit (vs. 15) addresses and lengths are adequate for a very long time
  - Therefore: Everything must be smaller than 16MB
  - Therefore: AMODE and RMODE were "patched in"
  - Therefore: no "scatter loading" by RMODE; entry points don't have own AMODE
- 8-character (vs. 6) upper-case EBCDIC names are adequate for a very long time
- Central storage is real (not virtual)
  - No page-outs of relocated pages

- Many current limitations that products must cope with:

  - Short names, 16MB size, mono-modal modules, rigid formats, inadequate ESD types, no room for descriptive data, internal table limits, strange loopholes, ...

- Some products invent "private" object formats, overload ESD names

  - Feed translator output through a "pre-linker" ahead of the Linkage Editor

```
            ┌──────────────────────────────┐
┌──────────┐│ ┌─────────┐    ┌─────────┐   │   ┌─────────┐   ┌─────────┐   ┌─────────┐
│          ││ │ Private │    │ Pre—    │   │   │ Normal  │   │ Link    │   │ Normal  │
│ Compiler │├─▶ Object  │──▶ │ Linker  │   ├──▶│ Object  │─▶ │ Editor  │─▶ │ Load    │
│          ││ │ Module  │    │         │   │   │ Module  │   │         │   │ Module  │
└──────────┘│ └─────────┘    └─────────┘   │   └─────────┘   └─────────┘   └─────────┘
            └──────────────────────────────┘
              Additional Linking Step
```

  - Updates may force complete re-link from private object modules

  - May have to "play games" to fool some existing tools (e.g. CMS TXTLIB)

- So, we must consider new formats for translator outputs

  - *Many* languages need more function: C, Ada, Fortran-90, anything O-O

- *Every* host program begins as an (unexecutable) object module

  – Must be transformed to an executable
    (MVS: load module; CMS: module; VSE: phase)

- OMs, LMs, LKED are at the heart of our business

  – Problems are pervasive, and affect <u>everyone</u>

  – Functional limitations also limit how we think about programs

- Some problems are generic, some are particular to each record type

- General problems:

  – Fixed format of records and fields

  – 16MB size/length limits due to 24-bit length and address fields

  – Inefficient use of file space

- ESD records:

  - Long names are impossible to accommodate (without loophole games)

  - 16 MB size/length limit on <u>everything</u>

  - Inadequate range of ESD types

  - Mono-modal modules and entry points

    — Entry points in a CSECT can't have different AMODEs

  - No properties information

    — Is it <u>really</u> RENT? Movable? REFR? REUS? Read-Only? Is it R/O data (constants)? Pure code? Code and R/W data?

  - No way to specify section alignment

  - CM/PR "ownership" very muddled, sometimes unknown

  - No data can be specified for CM items (must "merge" with an SD)

  - No attributes of modules or entries

    — Code? Data? (Should A or V point to this?)
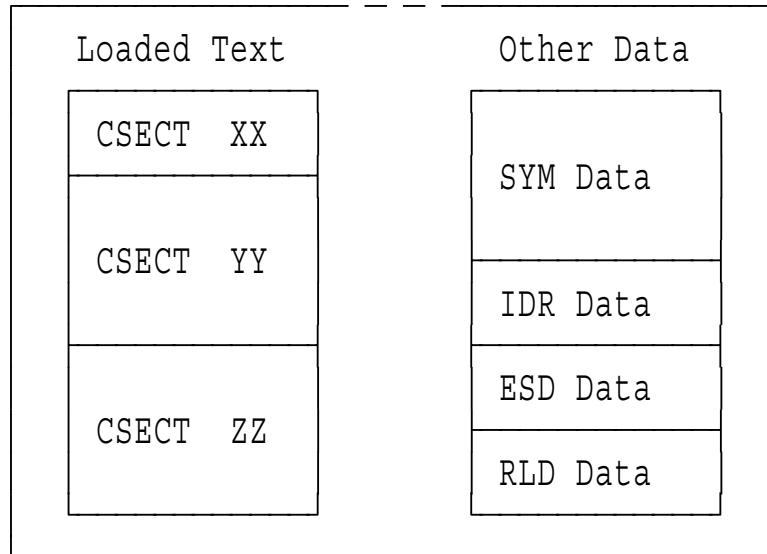
  - No way to provide descriptive data

- TXT records:

  - Maximum of 70% utilization

  - No way to specify text attributes

    — Is it code or data? Is it R/O? R/W? X/O?

    — Do pieces have different RMODEs?

  - Can't specify initializations for holes/gaps

  - Can't request data encoding or compression

- RLD records:

  - A-type and V-type adcons occasionally (mis-)used as equivalent

  - No checking is done between pointer/pointee

  - Cannot specify addressing modes for pointers

  - Cannot assign attributes for references

    — E.g. this is a pointer to data; to code; etc.

  - No "extended attributes" to allow interface-conformance checking

- SYM records:
  - Painfully complex, hard-to-use data formats
  - Symbolic names are truncated to 8 characters
  - No XREF and reference information is provided
  - No tie-backs from code and symbols to source statements
    — No source statements are retained, either!
    — Writing listing-scanners is not a very good approach...
  - SYM difficulties typically force users to choose beween
    — Putting symbols in TXT (giving bloated code)
    — Omitting symbols (giving harder debugging)
  - Basically, Assembler only...

- END records:
  - No way to specify entry point's AMODE
  - Cannot specify more than one deferred length
  - No provision for richer (and more useful) IDR data
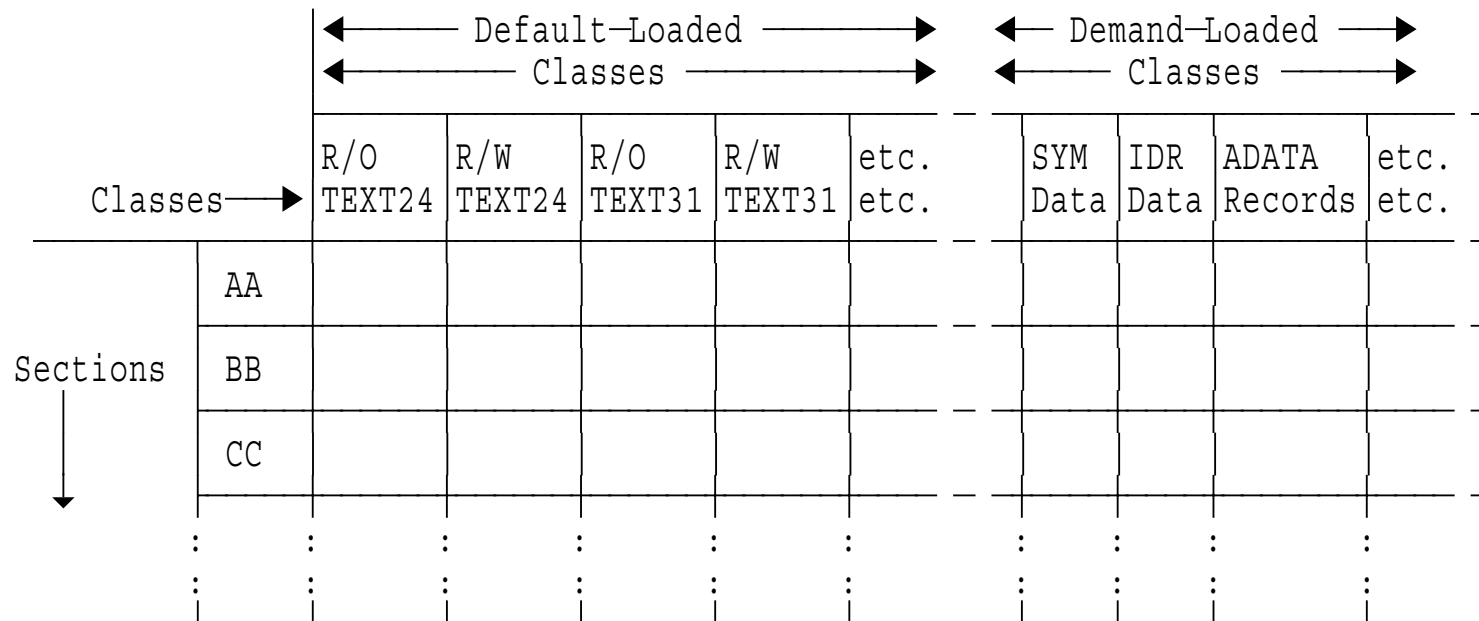 - Overloaded with too many miscellanous items

- And then there are Load Modules:

  - Inherit all the shortcomings of object modules

    — Short names, single modes, 16 MB limits, etc.

  - And add some new ones, too...

    — Peculiar module structures, inefficient record sizes

    — When re-linking, some items are "sticky"

      - PCs with code, CM lengths, PR length/alignment, SYM, IDR, ...

      - SYM records accumulate

    — System can't **LOAD** SYM, IDR data even if you want to!

  - Internal structures "exposed," impossible to change

- It's amazing that all this has worked (somehow) for so long a time!

- Linkage Editor is quirky, far too forgiving of errors, full of loopholes

  - Lots of "hand-crafted" bypasses (each detour is someone else's roadblock)

Looking Forward

- Totally new product and new technology

  - **Binder** replaces Linkage Editor, Batch Loader;
    Program Loader replaces Program Fetch

  - Upward compatible with previous products: support all functions/formats

  - Many new and enhanced capabilities

  - Answers a very large set of customer requirements

- Fixes a vast array of usability and performance problems

  - Many new messages, added information, and detailed diagnostics

  - Almost all internal constraints removed

- Supports **_Program Objects_** (a new form of load module "executable")

  - Enhances performance, flexibility, integrity

  - Internal structure not externalized; data-access interfaces provided

  - Stored in PDSE's, which fix almost all PDS problems (space, integrity, compression, performance, sharability, etc.)

- Supports a new **_Generalized Object File Format_**

- Base for all enhancements

```
        ┌ ─ ─ ─ ─ ─ ─ ┐
   ┌────────────────┬──────────────────┐
   │ Loaded Text    │ Other Data       │
   │  ┌──────────┐  │  ┌────────────┐  │
   │  │ CSECT XX │  │  │            │  │
   │  ├──────────┤  │  │ SYM Data   │  │
   │  │          │  │  │            │  │
   │  │ CSECT YY │  │  ├────────────┤  │
   │  │          │  │  │ IDR Data   │  │
   │  ├──────────┤  │  ├────────────┤  │
   │  │          │  │  │ ESD Data   │  │
   │  │ CSECT ZZ │  │  ├────────────┤  │
   │  │          │  │  │ RLD Data   │  │
   │  └──────────┘  │  └────────────┘  │
   └────────────────┴──────────────────┘
        └ ─ ─ ─ ─ ─ ─ ┘
```

- All of the "default-loaded" text has a single set of attributes

  – RMODE, AMODE, RENT or REUS, etc.

  – Effectively, a *single-component* module

- Other (not-loaded) module data not accessible via "normal" services

Classes →		Default-Loaded Classes					Demand-Loaded Classes			
		R/O TEXT24	R/W TEXT24	R/O TEXT31	R/W TEXT31	etc. etc.	SYM Data	IDR Data	ADATA Records	etc. etc.
	AA									
Sections ↓	BB									
	CC									
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

- Key new concept: independently-loaded *classes* of module data
  - All data in each class has identical attributes (e.g., RMODE)
  - Effectively, a *multi-component* module!
- Each *section* contributes to one or more classes
- Each *class* is analogous to a Load Module
- Other (not-loaded) module data accessible via Binder services

- Complete replacement for old Object Module

  – Supported in High Level Assembler, `C/C++`

- Variable-length or FB80 records

- Record types similar to OM types

  – Module Header (new)

  – External Symbol Definition: long names, richer set of types and attributes

  – Text: object code, other module-specific information (e.g., IDR)

  – Relocation Directory: relocation and other link-time actions

  – Deferred Section Length (formerly on OM END record)

  – End and Requested Entry

- Supports needs of "modern" languages

- Open-ended, flexible architecture

Summary

- Why program linking is a *Good Thing*

- What is in object modules, and where they come from

- How inter-module references are resolved to form an executable program

- What is in load modules, and how they are built by the Linkage Editor

- How load modules are loaded into storage and relocated

- Some history

- Where this technology is going

Note: many of these terms are used quite flexibly in this industry...

- Load, loading

  - Place a module into central storage

- Link, linking

  - Resolve symbolic (external) names into offsets or addresses

  - Combine multiple (input) name spaces into a single (output) name space

  - Sometimes called "binding" (but that term is much more general)

- Absolute loader

  - Places a module into storage at a fixed address, without relocating anything

  - Example: CMS's "traditional" non-relocatable MODULEs

- Relocate, relocation

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

- ## Relocating loader
  - Places modules into storage *and* updates (relocates) addresses to their actual "final" value
  - Example: Program Fetch, CMS Loader

- ## Linker, Linkage Editor, Binder
  - Creates linked relocatable modules for later loading
  - Example: Linkage Editor, DFSMS Binder

- ## Linking loader
  - Places modules into storage *with linking* immediately prior to program execution
  - Example: MVS Batch Loader, CMS's **LOAD ... (START...**

- ## Dynamic loading
  - Place modules into storage (with relocation) during program execution
  - Examples: portions of modules loaded by overlay, or modules loaded via LOAD, LINK, XCTL, ATTACH

- ## Dynamic linking
  - Place modules into storage *with linking* during program execution
  - Example: TSS

- 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

- Root segment

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

- Entry table

  – A special section inserted by the Linkage Editor at the end of an overlay segment, to assist branching into other segments

- Segment table

  – A special section created by the Linkage Editor describing segments of an overlay program, placed at the base of the root segment

- Reusable

    – Attribute of a module that describes the extent to which it can be reused or shared by multiple tasks in an address space

- Serially reusable

    – A module that can be executed by multiple tasks, one at a time in sequence

- Reenterable

    – A module that can be executed by multiple tasks concurrently

- Refreshable

    – A module that can be replaced by a new copy during execution without affecting its operation

1. OS/390 DFSMS Program Management (SC27-0806)

2. High Level Assembler for MVS & VM & VSE Language Reference (SC26-4940)

3. High Level Assembler for MVS & VM & VSE Programmer's Guide (SC26-4941)

   - These Assembler publications describe the most basic forms of language elements that create inputs to the Linkage Editor, Loader, and Binder.

4. *Linkers and Loaders*, by Leon Presser and John R. White, ACM Computing Surveys, Vol. 4 No. 3, Sept. 1972, pp. 149-167.

5. Linkage Editor and Loader User's Guide

6. Linkage Editor, Loader Program Logic manuals