Through LKED with Gun and Camera: Object/Load Modules, Link Editors, Loaders, and What They Do for You

SHARE 83, Session 4812

John R. Ehrman

IBM Santa Teresa Laboratory
555 Bailey Avenue, B16/D3
San Jose, CA 95141
1-408-463-3543
EHRMAN@VNET.IBM.COM or USIBM9WH@IBMMAIL[.COM]

10 Aug 94

Table of Contents

Introduction 1
Topic Overview 2
What Happens to Your Program? 3
Why is Linking Needed? 6
Putting the Pieces Back Together 7
Some General Definitions 9

Translator Output: Object Modules 11
Some IBM-Specific Definitions 12
Translator Output: The Object Module 13
Object Module External Symbol Dictionary (ESD) 14
Origins of External Symbol Dictionary Items 15
Object Module Machine-Language Text (TXT) 16
Object Module Relocation Dictionary (RLD) 17
Object Module Internal Symbol Dictionary (SYM) 18
Object Module End-of-Module (END) 19
Other Object Module Records (CMS) 20

Combining Object Modules with the Batch Loader 21
Combining Object Modules: a Simple Example 22
Combining Object Modules: First Object Module 23
Combining Object Modules: Second Object Module 24
Combining Object Modules: Batch Loader Actions 25

Table of Contents — Contents-2

Combining Object Modules: Resulting Program 26

Saving Linked Programs: Load Modules **27**
What and Why are Load Modules? 28
What Is In a Load Module? 29
Schematic of a "Normal" Load Module 30
Linkage Editor Inputs and Outputs 31
Linkage Editor Processing 32
Pseudo-Registers 33
Peculiarities of Load Modules 34

Overlay Modules **35**
What and Why are Overlays? 36
Example of an Overlay Structure 37
Arranging an Overlay Structure 38
An Overlay Structure In More Detail 39

Bringing Load Modules into Storage: Program Fetch **40**
Program Fetch -- A Relocating Loader 41

Looking Backward **42**
Some History 43
Assumptions and Constraints on 1963 Designs 44
Limitations and Extensions 45

Table of Contents — Contents-3

What Are The Problems? 46

Looking Forward **52**
The DFSMS/MVS Binder and Loader 53

Summary **54**
What We've Discussed 55
References 56

Linkage Editing, Loading, Object & Load Modules — SHARE 83, Session 4812
© IBM Corporation 1994
10 Aug 94

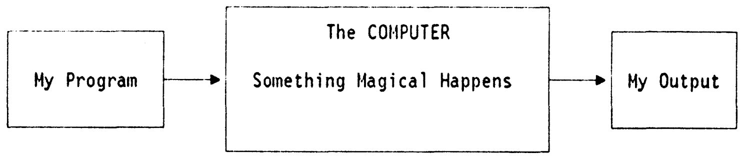
Introduction

Topic Overview

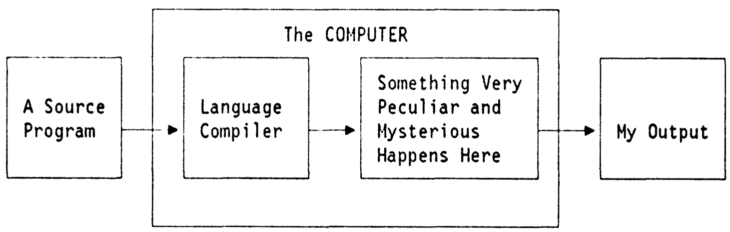
- What happens to programs "on the way to execution"
- Why program linking is needed
- What assemblers and compilers produce: object modules
- What program linking does with object modules
- Saving the results of linking: load modules
- What happens when load modules are put into storage
- Why the Linkage Editor and Loader are the way they are
- The future: the good things the new Binder does for you

What Happens to Your Program? 3

1. The Beginner's View



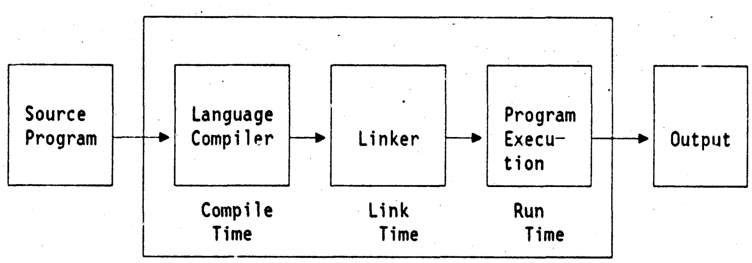
2. The After-a-Little-Experience View



- We learn to distinguish between compile time and run time

What Happens to Your Program? ... 4

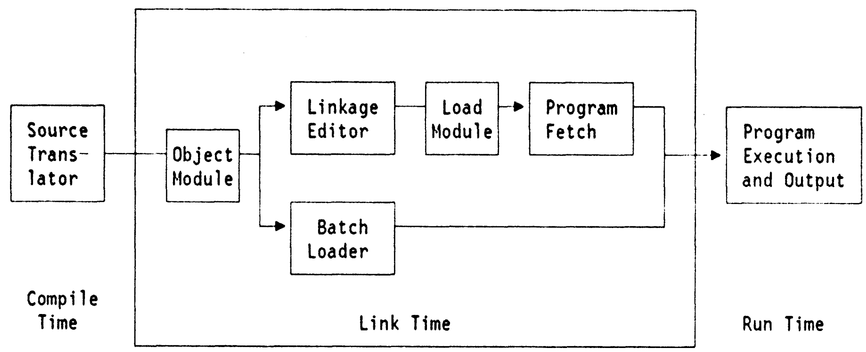
3. The After-Some-More-Experience View



- We learn to distinguish among compile, link, and run times

What Happens to Your Program? ... 5

4. Our View



Our Concerns: The Program Linking Process

- Our focus will be almost entirely on the five items in the "Link Time" box
- We will refer to some compile-time and run-time topics and issues

Why is Linking Needed? 6

- Anything that gets "big" is hard to manage
- The world's oldest paradigm for handling big 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

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812
© IBM Corporation 1994 10 Aug 94

Putting the Pieces Back Together 7

- Putting the pieces back together ("binding") can occur at many times
  - Compile time -- compile all needed items from source
  - Link Edit (pre-execution) time -- everything "bound" prior to execution
  - Program initiation time -- everything "bound" immediately prior to 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

Putting the Pieces Back Together ... 8

- 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**: let the pieces refer to one another
- 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?
  - DFSMS/MVS Binder is much more informative (more about this, later)

Linkage Editing, Loading, Object & Load Modules
SHARE 83, Session 4812
© IBM Corporation 1994
10 Aug 94

Some General Definitions 9

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

Some General Definitions ... 10

- Relocating loader
  - Places modules into storage ***and*** updates (relocates) addresses to their actual "final" value
  - Example: Program Fetch, CMS Loader
- Linker, Linkage Editor
  - Creates linked relocatable modules for later loading
  - Example: Linkage Editor
- Linking loader
  - Places modules into storage ***with linking*** immediately prior to program execution
  - Example: MVS Batch Loader
- Dynamic loading
  - Place modules into storage (with relocation) during program execution
  - Examples: parts 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

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812
© IBM Corporation 1994 10 Aug 94

Translator Output: Object Modules

- For the exciting details, see Appendix C of SC26-4941,

  High Level Assembler/MVS & VM & VSE Programmer's Guide

Some IBM-Specific Definitions

- Control Section (**CSECT**, for short)
  - A collection of program elements, all 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 will be placed during program relocation
- Pseudo-Register (or, External Dummy Section)
  - A special type of external symbol with a separate "name space"
  - More about these, later

Translator Output: The Object Module — 13

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

ESD	External Symbol Dictionary
TXT	Machine Language instructions and data ("Text")
RLD	Relocation Dictionary
SYM	Internal Symbols
END	End of Object Module, with **IDR** (Identification Record) data

- One object module per Compilation Unit
- "Batch" translations may produce multiple object modules

Object Module External Symbol Dictionary (ESD) — 14

- 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 classes of external symbol:

  SD,CM Section Definition: the name of a control section
  - Data: ESDID, length, section-origin address, AMODE & RMODE
  - Blank **SD** name sometimes called Private Code (**PC**)
  - Common (**CM**) handled differently from **SD** items

  LD Label Definition: the name of a position at a fixed offset within a Control Section; typically, an Entry Point
  - Data: Address of the label, and ESDID of the section it's in

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

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

- ESD records must appear first in each object module

Origins of External Symbol Dictionary Items 15

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

ESD item	Assembler	VS Fortran	OS PL/I	VS COB. II	C/370
SD	Csect, Rsect	Routine, Block Data	Procedure	Outermost program	R/W data
CM	Com	Common	External static		
ER	Extrn, V-con	Call, Common	Call, data reference	Static Call Literal	Call, data reference
LD	Entry	Entry	Entry	Entry	Function
PR, XD	DXD, Q-con + Dsect		File, Fetchable, Controlled		Writable static
WX	Wxtrn				

Object Module Machine-Language Text (TXT) 16

- 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

Object Module Relocation Dictionary (RLD) — 17

- Packed stream of 2-byte or 4-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" it should contain
     - I.e., what it points to
  2. **P Pointer**: ESDID of the section where the constant resides
     - I.e., where to find it
  3. **Address**: the address at which the constant resides within its section (as specified by the P pointer)
  4. **Length**: the constant's length (in bytes)
  5. **Type**: whether it's an A-type (data), V-type (branch), Q-type (PR offset), or CXD (PR "Cumulative Length")

     Warning!! A- and V-type constants can be *very* different!! (More later...)
  6. **Direction**: whether the target address should be added or subtracted for A-type constants

Object Module Internal Symbol Dictionary (SYM) — 18

- Contains **internal symbols** used by source translator
  - Produced by Assembler, VS Fortran
- SYM information is (sometimes) useful for debugging
- Ghastly bit-squeezing packed format (details are truly impressive)
  - Maximum symbol length is 8 characters
- 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 format

Object Module End-of-Module (END) 19

- Primary function is to signal the end of the object module
- Some additional (optional) information may be provided:
  - Requested execution-time entry point
    - By ESDID and address, or by external name
    - These requests may be overridden by other factors or controls
  - Actual length of a Control Section whose length was not specified on its ESD record
    - This feature saves effort in some compilers
  - 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

Other Object Module Records (CMS) 20

- 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
- It can do a lot more than this example shows

Combining Object Modules: a Simple Example

- 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(ZDATA)
    |          - - -
    |          ENTRY ZDATA
260 | ZDATA
    |          - - -
```

```
         Module 2
Loc
000 | SUB
    |          COMMON /WORK/ ...
    |          EXTERNAL ZDATA
    |          - - -
700 |          Addr(WORK)
704 |          Addr(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

Combining Object Modules: First Object Module 23

- 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=260        '01234567890'   Text for MAIN, address 260
TXT    ID=1 ...              etc.           Text for MAIN
RLD    PID=1 RID=3 Addr=200 Len=4 Dir=+     RLD item for Addr(SUB)
RLD    PID=1 RID=2 Addr=204 Len=4 Dir=+     RLD item for Addr(WORK)
RLD    PID=1 RID=1 Addr=208 Len=4 Dir=+     RLD item for Addr(ZDATA)
END    Entry=MAIN                           End of module, entryname=MAIN
```

- ESD defines two control sections (MAIN and WORK), one entry (ZDATA), one external reference (SUB)
- RLD contains information about three address constants

Combining Object Modules: Second Object Module 24

- 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 Dir=+     RLD item for Addr(WORK)
RLD    PID=1 RID=3 Addr=704 Len=4 Dir=+     RLD item for Addr(ZDATA)
END                                         End of module
```

- ESD defines two control sections (SUB and WORK), one external reference (ZDATA)
- RLD contains information about two address constants

Combining Object Modules: Batch Loader Actions 25

- The Batch Loader
  1. Builds a single ("Composite") ESD
     - Merges ESD information from the object modules
     - Renumbers ESDIDs, assigns adjusted address values to all symbols
  2. Places text in storage at designated addresses
  3. Determines length of COMMON (retains longest length)
     - Allocates storage for it
  4. Relocates address constants
  5. Sets entry point address
  6. Enters loaded program
- Suppose initial program load address is 123500

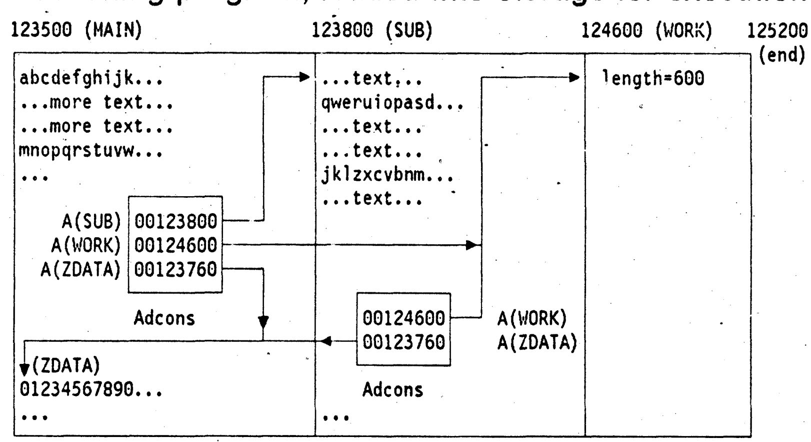
- Composite ESD

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)

Combining Object Modules: Resulting Program 26

- The resulting program, loaded into storage for execution:



- Storage is allocated for three control sections (two SD, one CM)
- Address constants are resolved to designated addresses
- Entry point is at address of MAIN (123500)

Saving Linked Programs: Load Modules

What and Why are Load Modules?

- Basic executable unit for MVS-like systems
  - The world's longest-surviving form of "executable binary"
- Designed 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

What Is In a Load Module? 29

- Load module 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

- Additional items having no object-module analogs

CTL	Control records, for reading and relocating text records
SEGTAB	Segment table, for overlay structures
ENTAB	Entry table, for overlay structures
EOS	End of Segment, for overlay structures

Schematic of a "Normal" Load Module 30

- Basic format is called "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 (TTR)

Linkage Editor Inputs and Outputs 31

- 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

Linkage Editor Processing 32

- Two-pass process (very much like an assembler!)
- Pass 1
  - Read all inputs (explicitly or implicitly designated)
    - If not NCAL, unresolved ERs cause library search (WXs never do)
  - 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.
- Intermediate processing
  - Resolve interdependences
  - Assign relative addresses
  - 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
  - STOW directory entry (or entries, if ALIASes)
  - Write XREF (if module didn't fit in storage)

Pseudo-Registers 33

- Allow sharing by name in separately translated re-entrant programs
- 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" (PRV; up to 1024 more "registers")
- Resolved somewhat like commons:
  - But: no storage allocated at link time, as for commons
  - If multiple definitions, longest length and strictest alignment win
  - Accumulated length/alignment of PRV items then determine offsets
  - Offset value placed in Q-type address constants referencing PR name
  - Total size of the "link-time DSECT" is placed in a "CXD" adcon item
- Runtime code must allocate a storage area of the correct right (CXD) size
- Runtime references access fields at desired offsets in that area, using the Q-con contents for "displacements"

Peculiarities of Load Modules 34

- 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 sizes
  - SYM ≤ 244; CESD ≤ 248; IDR, CTL, RLD ≤ 256; Text ≤ track length
- 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
    - Alice would be at home in *this* Wonderland!
  - TSS Linkage Editor didn't allow this confusion

Overlay Modules

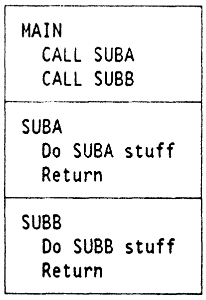
What and Why are Overlays?

- Overlays are more complex than block-format modules
  - Different parts of a module may share the same storage
    - At different times, of course!
  - Require special Linkage Editor considerations
- Pros:
  - Faster initiation: only part of the program need be loaded to start
  - Economical storage use: only load what's needed, when it's needed
  - Can always re-link to block format if there's enough storage
    - ***But:*** Behavior may be different, due to loss of re-initializations!
- Cons:
  - Programs cannot be shared (no re-enterability)
  - More complex to specify, greater care needed in coding
    - Local data may or may not be "persistent" across calls
    - Distinction between V-type and A-type adcons is important
    - External data sharing may be more complicated
  - Additional overhead in calls to segments needing to be loaded
  - Calls among certain modules may be forbidden (or wrong)

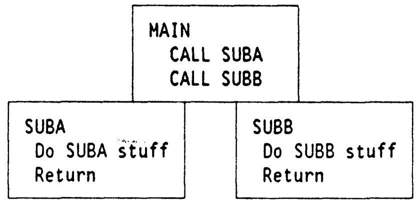
Example of an Overlay Structure

37

- Suppose MAIN calls SUBA and SUBB
  - Neither calls the other
- In “block format,” they would appear in storage as



- SUBA and SUBB might be overlaid, like this:

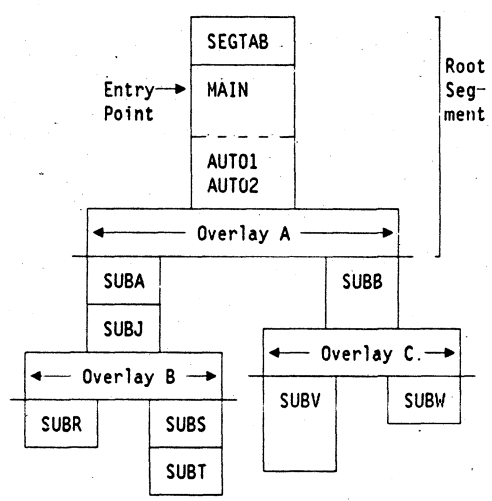


- SUBA and SUBB share the same storage
- The overlay supervisor must (help) make this work!

Arranging an Overlay Structure

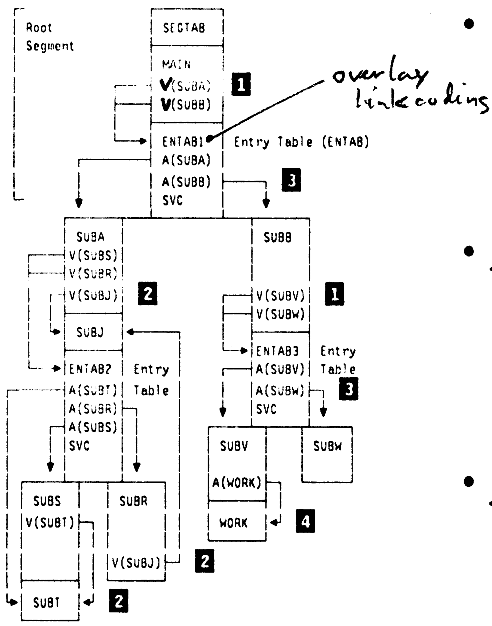
38

- Figure out what modules can share storage
- Draw an “overlay tree” of the structure
  - Structured as a tree, with root at top (low address)
  - 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)



An Overlay Structure In More Detail

39



- Each segment with subsidiary segments is suffixed with an Entry Table to assist loading of the "lower" segments
  - SVC instructions 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
  - Addresses in ENTAB resolve directly to SUBs (**3**)
  - Sections in same segment (**4**)

40

Bringing Load Modules into Storage: Program Fetch

Program Fetch -- A Relocating Loader

41

- 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 have RLD information for preceding text block
- A,V-cons relocated using only **address** information in RLD
  - R and P pointers ignored; RLD information discarded after relocation
  - Q-cons and CXD were completed at linkage-edit time
- Note: two levels of relocation are involved:
  1. Linkage Editor adjusts addresses relative to module zero origin
  2. Program Fetch adjusts addresses relative to module's "load address"
- Overlay Supervisor
  - SEGTAB and ENTABs manage segment traffic; Program Fetch loads segments

42

Looking Backward

Some History 43

- Linkage Editor
  - Written in 1963-65 by small team in IBM Poughkeepsie
  - 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 (32KB machines were ***big!***)
- OS Batch Loader
  - Written much later (about 1972)
  - Appeared with OS/360 Release 17
- Very advanced technology for that time
  - Long ago, in a far away galaxy, ...

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812

10 Aug 94

Assumptions and Constraints on 1963 Designs 44

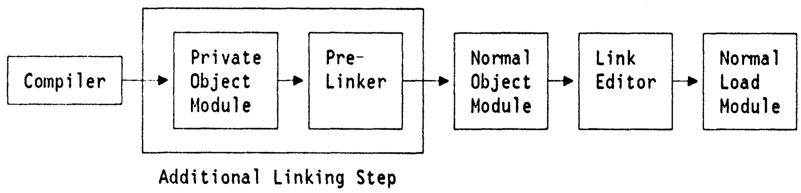
- Early-binding philosophy: systems are expensive, people are cheap
  - Programs run for long periods between needed changes
  - Therefore: recompile "deltas" and re-link them into the application module
- Re-linking is cheaper than re-building from scratch
  - Therefore: keep enough info within the 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 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 upper-case EBCDIC names are adequate for a very long time
- Central storage is real (not virtual)

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812

10 Aug 94

Limitations and Extensions 45

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



  - Updates may force complete re-link from private objects
  - May have to "play games" to fool some existing tools (e.g. CMS TXTLIB)
- We must consider new formats for translator outputs
  - *Many* languages need more function: C, Ada, Fortran-90, anything O-O

What Are The Problems? 46

- One hardly knows where to start!
- 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

Linkage Editing, Loading, Object & Load Modules
SHARE 83, Session 4812
© IBM Corporation 1994
10 Aug 94

What Are The Problems? ... 47

- ESD records:
  - Long names are impossible to accommodate (without loophole games)
  - 16 MB size/length limit on everything
  - 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 really 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
  - No data can be specified for CM items
  - No attributes of modules or entries
    - Code? Data? (Should A or V point to this?)
  - No way to provide descriptive data

What Are The Problems? ... 48

- TXT records:
  - Maximum of 70% utilization
  - No way to specify text attributes
    - Is it code/data? Is it RO/RW/XO?
    - Do pieces have different RMODEs?
  - Can't specify initializations for holes/gaps
  - Can't request data encoding or compression

What Are The Problems? ... 49

- RLD records:
  - Available "type info" is often abused (or not respected) by coders
    - A-type and V-type adcons (mis-)used as essentially 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

What Are The Problems? ... 50

- 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...
- 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

Linkage Editing, Loading, Object & Load Modules
SHARE 83, Session 4812
© IBM Corporation 1994
10 Aug 94

What Are The Problems? ... 51

- 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, ...
    - System can't **LOAD** SYM, IDR data even if you want to!
- It's amazing that all this has worked (somehow) for so long a time!

52

Looking Forward

The DFSMS/MVS Binder and Loader

53

- Totally new product and new technology
  - **Binder** replaces Linkage Editor, Batch Loader
  - **Loader** replaces Program Fetch
  - 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
  - Linkage Editor is quirky, far too forgiving of errors, full of loopholes
- Supports a new "execution unit" – a ***Program Object***
  - 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.)
- Base for future enhancements
- Available March 1993, in DFSMS/MVS 1.1

Linkage Editing, Loading, Object & Load Modules

SHARE 83, Session 4812
10 Aug 94

54

Summary

Linkage Editing, Loading, Object & Load Modules
© IBM Corporation 1994
SHARE 83, Session 4812
10 Aug 94

What We've Discussed 55

- 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

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812
 10 Aug 94

References 56

1. DFSMS/MVS V1R1 Program Management (SC26-4916)
2. Linkage Editor and Loader User's Guide
3. Linkage Editor, Loader Program Logic manuals
4. High Level Assembler/MVS & VM & VSE Language Reference (SC26-4940)
5. High Level Assembler/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.
6. *Linkers and Loaders*, by Leon Presser and John R. White, ACM Computing Surveys, Vol. 4 No. 3, Sept. 1972, pp. 149-167.

Linkage Editing, Loading, Object & Load Modules SHARE 83, Session 4812
 Rev. 20 Jun 94, 1630 10 Aug 94