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
© IBM Corporation 1994

10 Aug 94

Table of Contents

Contents-1

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 .................................
  Some IBM-Specific Definitions ............................... 11
  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>
<thead>
<tr>
<th>Table of Contents</th>
<th>Contents-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combining Object Modules: Resulting Program</td>
<td>26</td>
</tr>
<tr>
<td>Saving Linked Programs: Load Modules</td>
<td>27</td>
</tr>
<tr>
<td>What and Why are Load Modules?</td>
<td>28</td>
</tr>
<tr>
<td>What Is In a Load Module?</td>
<td>29</td>
</tr>
<tr>
<td>Schematic of a “Normal” Load Module</td>
<td>30</td>
</tr>
<tr>
<td>Linkage Editor Inputs and Outputs</td>
<td>31</td>
</tr>
<tr>
<td>Linkage Editor Processing</td>
<td>32</td>
</tr>
<tr>
<td>Pseudo-Registers</td>
<td>33</td>
</tr>
<tr>
<td>Peculiarities of Load Modules</td>
<td>34</td>
</tr>
<tr>
<td>Overlay Modules</td>
<td>35</td>
</tr>
<tr>
<td>What and Why are Overlays?</td>
<td>36</td>
</tr>
<tr>
<td>Example of an Overlay Structure</td>
<td>37</td>
</tr>
<tr>
<td>Arranging an Overlay Structure</td>
<td>38</td>
</tr>
<tr>
<td>An Overlay Structure In More Detail</td>
<td>39</td>
</tr>
<tr>
<td>Bringing Load Modules into Storage: Program Fetch</td>
<td>40</td>
</tr>
<tr>
<td>Program Fetch -- A Relocating Loader</td>
<td>41</td>
</tr>
<tr>
<td>Looking Backward</td>
<td>42</td>
</tr>
<tr>
<td>Some History</td>
<td>43</td>
</tr>
<tr>
<td>Assumptions and Constraints on 1963 Designs</td>
<td>44</td>
</tr>
<tr>
<td>Limitations and Extensions</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Contents-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Are The Problems?</td>
<td>46</td>
</tr>
<tr>
<td>Looking Forward</td>
<td>52</td>
</tr>
<tr>
<td>The DFSMS/MVS Binder and Loader</td>
<td>53</td>
</tr>
<tr>
<td>Summary</td>
<td>54</td>
</tr>
<tr>
<td>What We’ve Discussed</td>
<td>55</td>
</tr>
<tr>
<td>References</td>
<td>58</td>
</tr>
</tbody>
</table>
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?

1. The Beginner's View

```
My Program → The COMPUTER
Something Magical Happens → My Output
```

2. The After-a-Little-Experience View

```
A Source Program
→ Language Compiler
Something Very Peculiar and Mysterious Happens Here
→ My Output
```

- We learn to distinguish between compile time and run time

3. The After-Some-More-Experience View

```
Source Program → Language Compiler → Linker → Program Execution → Output
Compile Time
Link Time
Run Time
```

- We learn to distinguish among compile, link, and run times
4. Our View

- 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?

- 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
Putting the Pieces Back Together

- 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 ...

- 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)
Some General Definitions

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**
  - 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
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

- 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)

- 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

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

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

Object Module Machine-Language Text (TXT)

- 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)

- 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)

- 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)

- Primary function is to signal the end of the object module
- Some additional (optional) information may be provided:
  - Requested execution-time entry point
    - By ESID 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)

- 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:

<table>
<thead>
<tr>
<th>Module 1</th>
<th>Module 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc</td>
<td>Loc</td>
</tr>
<tr>
<td>000 MAIN</td>
<td>000 SUB</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  | COMMON /WORK/ ... | COMMON /WORK/ ...
  | CALL SUB | EXTERNAL ZDATA |
  | Addr(SUB) | - - - |
  | Addr(WORK) | Addr(WORK) |
  | Addr(ZDATA) | Addr(ZDATA) |
  | ENTRY ZDATA | - - - |
  | 260 ZDATA | 700 704 |

  (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

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

<table>
<thead>
<tr>
<th>ESD</th>
<th>ID=1</th>
<th>MAIN</th>
<th>Addr=000</th>
<th>Len=300</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESD</td>
<td>ID=2</td>
<td>WORK</td>
<td>Addr=000</td>
<td>Len=600</td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESD</td>
<td>ID=1</td>
<td>ZDATA Addr</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESD</td>
<td>ID=3</td>
<td>SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>Addr=000</td>
<td>'abcdefgijk'</td>
<td>etc.</td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>...)</td>
<td>etc.</td>
<td></td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>Addr=100</td>
<td>'mnopqrstuvwxyz'</td>
<td>etc.</td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>Addr=260</td>
<td>'01234567890'</td>
<td>etc.</td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>...)</td>
<td>etc.</td>
<td></td>
</tr>
<tr>
<td>RLD</td>
<td>PID=1</td>
<td>RID=3</td>
<td>Addr=700</td>
<td>Len=4</td>
</tr>
<tr>
<td>RLD</td>
<td>PID=1</td>
<td>RID=2</td>
<td>Addr=204</td>
<td>Len=4</td>
</tr>
<tr>
<td>END</td>
<td>Entry=MAIN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

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

<table>
<thead>
<tr>
<th>ESD</th>
<th>ID=1</th>
<th>SUB</th>
<th>Addr=000</th>
<th>Len=800</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESD</td>
<td>ID=2</td>
<td>WORK</td>
<td>Addr=000</td>
<td>Len=400</td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESD</td>
<td>ID=3</td>
<td>ZDATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>Addr=040</td>
<td>'qweruiopasd'</td>
<td>etc.</td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>...)</td>
<td>etc.</td>
<td></td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>Addr=180</td>
<td>'jklxcvbmn'</td>
<td>etc.</td>
</tr>
<tr>
<td>TXT</td>
<td>ID=1</td>
<td>...)</td>
<td>etc.</td>
<td></td>
</tr>
<tr>
<td>RLD</td>
<td>PID=1</td>
<td>RID=2</td>
<td>Addr=700</td>
<td>Len=4</td>
</tr>
<tr>
<td>RLD</td>
<td>PID=1</td>
<td>RID=3</td>
<td>Addr=204</td>
<td>Len=4</td>
</tr>
<tr>
<td>END</td>
<td>Entry=MAIN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

- **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

- **Composite ESD**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>ESDID</th>
<th>Addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>SD</td>
<td>01</td>
<td>123500</td>
<td>300</td>
</tr>
<tr>
<td>ZDATA</td>
<td>LD</td>
<td>01</td>
<td>122760</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>SD</td>
<td>02</td>
<td>123800</td>
<td>800</td>
</tr>
<tr>
<td>WORK</td>
<td>CH</td>
<td>03</td>
<td>124600</td>
<td>600</td>
</tr>
<tr>
<td>(end)</td>
<td></td>
<td></td>
<td></td>
<td>125200</td>
</tr>
<tr>
<td>entry</td>
<td></td>
<td>01</td>
<td>123500</td>
<td></td>
</tr>
</tbody>
</table>

(For this example, values are given in decimal)

- **Suppose initial program load address is 123500**

Combining Object Modules: Resulting Program

- **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)
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?

- 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

- 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 Info about first text record
  - CTL First text record, placed at module load address
  - Text Relocate first text record, and read second text record
       at specified offset from module load address
  - CTL/RLD Last text record
  - RLD Relocate last text record, signal end of module

* Location of first text record kept in PDS directory \( TTR \)
Linkage Editor Inputs and Outputs

• 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

• 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

- 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 right (CXD) size
- Runtime references access fields at desired offsets in that area, using the Q-con contents for "displacements"

### Peculiarities of Load Modules

- 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 $\leq$ 244; CESD $\leq$ 248; IDR, CTL, RLD $\leq$ 256; Text $\leq$ 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
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

- Suppose MAIN calls SUBA and SUBB
  - Neither 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
  Do SUBA stuff
  Return

SUBB
  Do SUBB stuff
  Return
```

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

Arranging an Overlay Structure

- 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

- 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)

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

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

- 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

Looking Backward
**Some History**

- **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, ...

---

**Assumptions and Constraints on 1963 Designs**

- **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)**
Limitations and Extensions

- 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

  Additional Linking Step

  - 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?

- 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
What Are The Problems? ...

- **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
    - 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

- **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?

- 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

- 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
What Are The Problems? ...

• 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!

Looking Forward
The DFSMS/MVS Binder and Loader

- 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

Summary
What We’ve Discussed

• 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

References

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