Enterprise COBOL Debugging and Maintenance

Enterprise COBOL Debugging and Maintenance - Course Objectives

On successful completion of this course, the student, with the aid of the appropriate reference materials, should be able to:

1. Describe the general structure of the LE program management model
2. Describe the outputs of the IBM Enterprise COBOL compiler, and use these outputs correctly in problem determination and dump debugging
3. Approach debugging in an orderly, efficient fashion
4. Locate data items from a COBOL program in an LE CEEDUMP
5. Better understand subroutines and parameters in a COBOL environment
6. Use the Program Binder to maintain load modules and program objects
7. Understand LE debugging facilities such as condition handling and the CEE3DMP, CEE3ABD, and CEE2AB2 LE services
8. Use the appropriate COBOL compiler debugging techniques to assist in tracking down and solving errors
9. Use appropriate Binder options and control statements, including creating a program object with a segment below the line and a segment above the line.

Enterprise COBOL Debugging and Maintenance - Topical Outline

Day One

Day One, continued

Day Two

Day Two, continued

Section Preview

- **Language Environment - An Introduction**
  - **What is Language Environment?**
  - **LE Conforming Programs**
  - **LE Services**
  - **Using LE Services**
  - **Invoking LE Services**
  - **The LE Run-Time Environment**
  - **LE Program Management**

What Is Language Environment?

- **Language Environment (LE) is a set of programs that provide the following capabilities**
  - **A common run-time environment for many languages**
    - ✗ COBOL, PL/I, C/C++, Assembler, FORTRAN, Java
  - **A set of callable routines that provide useful services for applications written using LE conforming compilers**
    - ✗ Date and time, storage management, mathematical, etc.
  - **LE is used to support z/OS UNIX System Services, also called, more simply, z/OS UNIX**
    - ✗ Including support for Unix file structures (directories, subdirectories, ... , files) and standard UNIX calls and services
- **Initially, LE was an option of the operating system - now, LE comes with the operating system and must always be available**

LE Conforming Programs

- A program is “LE conforming” if it establishes or runs under the LE run-time environment and follows LE conventions

- Programs compiled using the compilers designed for the LE environment are automatically LE conforming:
  - IBM Enterprise COBOL for z/OS, COBOL for OS/390 & VM
  - IBM Enterprise PL/I for z/OS, PL/I for MVS & VM
  - XL C/C++, z/OS C/C++

- These compilers automatically generate dynamic calls to the Language Environment initialization routines
  - In fact, programs compiled and linked using these compilers must run in the LE environment

- Of course, Assembler programs can also be written to invoke the LE initialization routines, but the Assembler doesn't automatically generate the linkages to these routines

LE Services

❒ **As an overview, the services available to Language Environment conforming programs fall into the following categories**

- ♦ **Storage Management - obtain and free memory dynamically**

- ♦ **Condition Handling - detect errors and other conditions, and handle conditions in a consistent manner**

- ♦ **Messaging Services - define message files that can be shared by many programs; issue messages, including**
  - ✗ substituting variables, from programs;
  - ✗ route messages to various target locations

- ♦ **Date and Time Services - get and store date and time in various formats; convert between formats**

- ♦ **Debugging Services - retrieve / set error codes; generate dumps; invoke a debug tool**

- ♦ **Mathematical Services - Trigonometric functions; exponential and logarithmic functions; etc.**

- ♦ **International Services - retrieve / set country, language, currency, and similar attributes, including support for locales**

Using LE Services

- **Language Environment services are accessed using CALL statements (or CALL-like mechanisms, such as function references in C/C++)**
  - **All Language Environment services are subroutines**
  - **All these subroutine names begin with “CEE”**
- **A program using Language Environment services must be compiled using the appropriate compilers**
  - **Just inserting CALLs to these services and then compiling with an earlier compiler won't work because the service calls assume the LE environment has been established**
- **However, note that non-LE conforming programs can run in the LE environment (a COBOL II load module, for example, can be called by an Enterprise COBOL main program)**

Invoking LE Services

- **COBOL programs**
  - **Standard CALL syntax applies to invoking services, for example**

```
Call 'CEEMSG' using in-token, dest2, fc-token
```

  - **On return, check "fc-token", not RETURN-CODE**
  - **Calls may be either static or dynamic**

- **The fc-token field is a 12-byte field that returns detailed information on how the request went**
  - **Details beyond the scope of this course, but sufficient to:**
    - ✗ Set to low-values before requesting service
    - ✗ Check for low-values after return from service
      - ➢ If not still low-values then some kind of error occurred

The LE Run-time Environment

❒ **To understand debugging in the LE environment, there are a number of issues we need to discuss**

- ♦ **The LE program management model**
  - ✗ Basically, LE hides the traditional MVS and z/OS program management structure, introducing terms like Process, Enclave, and Thread
- ♦ **LE condition handling**
  - ✗ LE provides services available to the application programmer for detecting and handling conditions
  - ✗ And, if the user doesn't use these facilities, LE will
- ♦ **LE Dumps**
  - ✗ The layout for, and information in, an LE dump is based on the program management model and the condition handling facilities of LE
    - ➢ LE writes dumps to a data set with a DDname of CEEDUMP instead of the Abend dump data set SYSUDUMP
    - ➢ If you don't provide a CEEDUMP statement, LE will dynamically allocate one if it needs to create a dump

❒ **So we begin this part of our odyssey with a brief look at the LE program management model ...**

LE Program Management

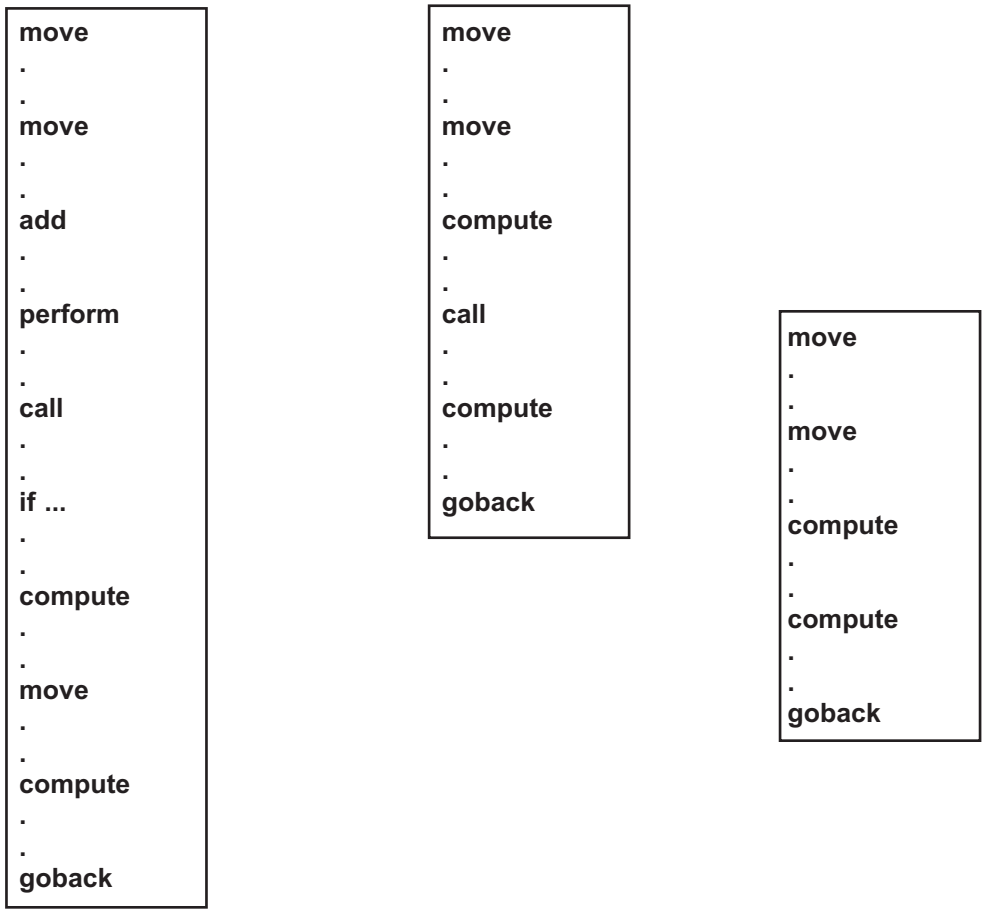
- **Language Environment manages programs and resources using a model that recognizes**
  - **Thread - the execution of an application's program(s); think 'task' in traditional z/OS terms**
  - **Enclave - programs and storage used by one or more related threads; an enclave consists of: a single main program, any number of sub-programs (subroutines), and storage shared among the programs; think 'run-unit'**
  - **Process - one or more related enclaves and their shared resources: a message file and the runtime library (for batch, think: a logical chunk of an address space containing related programs, data, and control blocks; for online programs, think: transaction)**
- **When you run an LE main program (LE-conforming Assembler or LE-compliant high level language compiler), LE initializes the run-time environment (process) by initializing an enclave and an initial thread**
  - **Enclave initialization acquires an initial heap storage and establishes the starting values of attributes such as the country and language settings and the century window**
  - **Thread initialization acquires a stack, enables a condition manager, and launches the main program**
- **You can modify initialization by running a user exit**

LE Program Management, continued

- Let's examine this program management model a little more closely

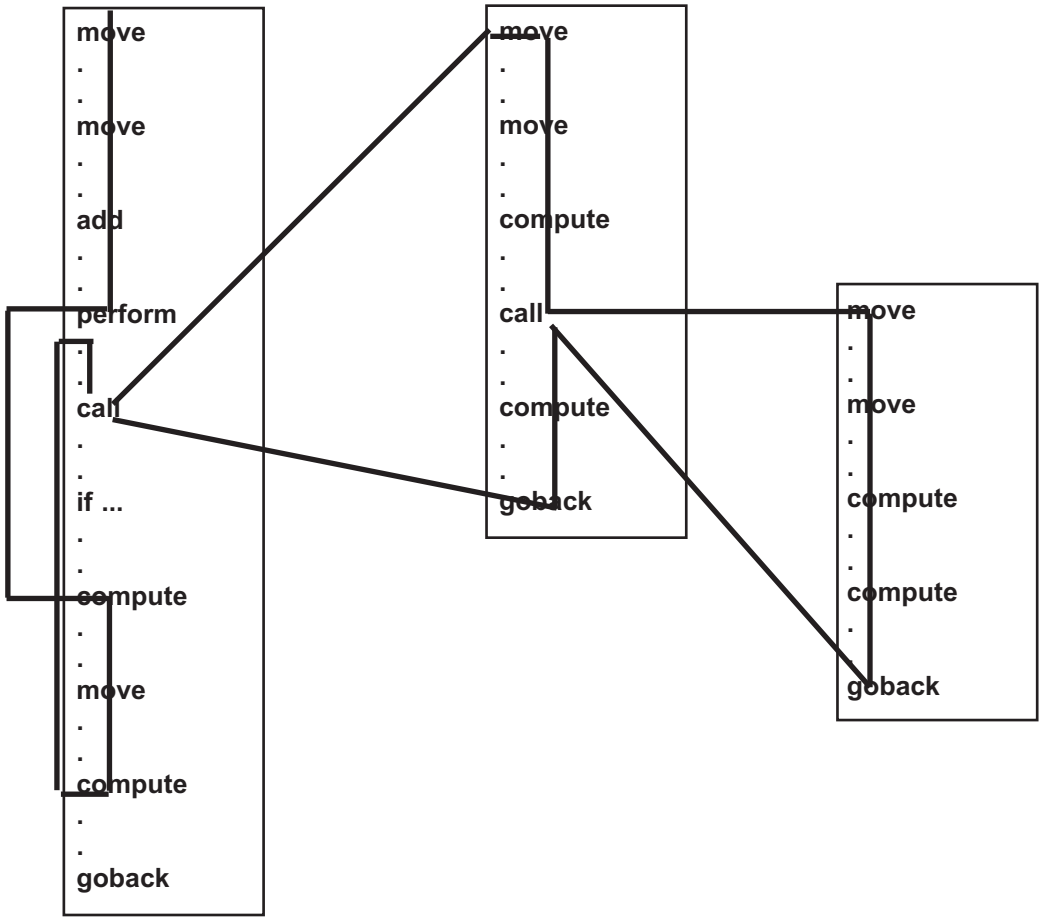
- Start with the enclave: this is really the most familiar concept for most programmers:
  - A mainline and the subroutines it calls (including subroutines called by subroutines, etc.)
  - The subroutines may be called statically or dynamically

- An enclave



LE Program Management, continued

❒ **Now, as the program executes, if we could trace its progress we might see a line of execution something like this:**



❒ **This line of instruction execution is called a thread**

- **Note that although there are three programs here, there is a single thread**

LE Program Management, continued

❒ **Finally, the overall umbrella in LE is the Process**

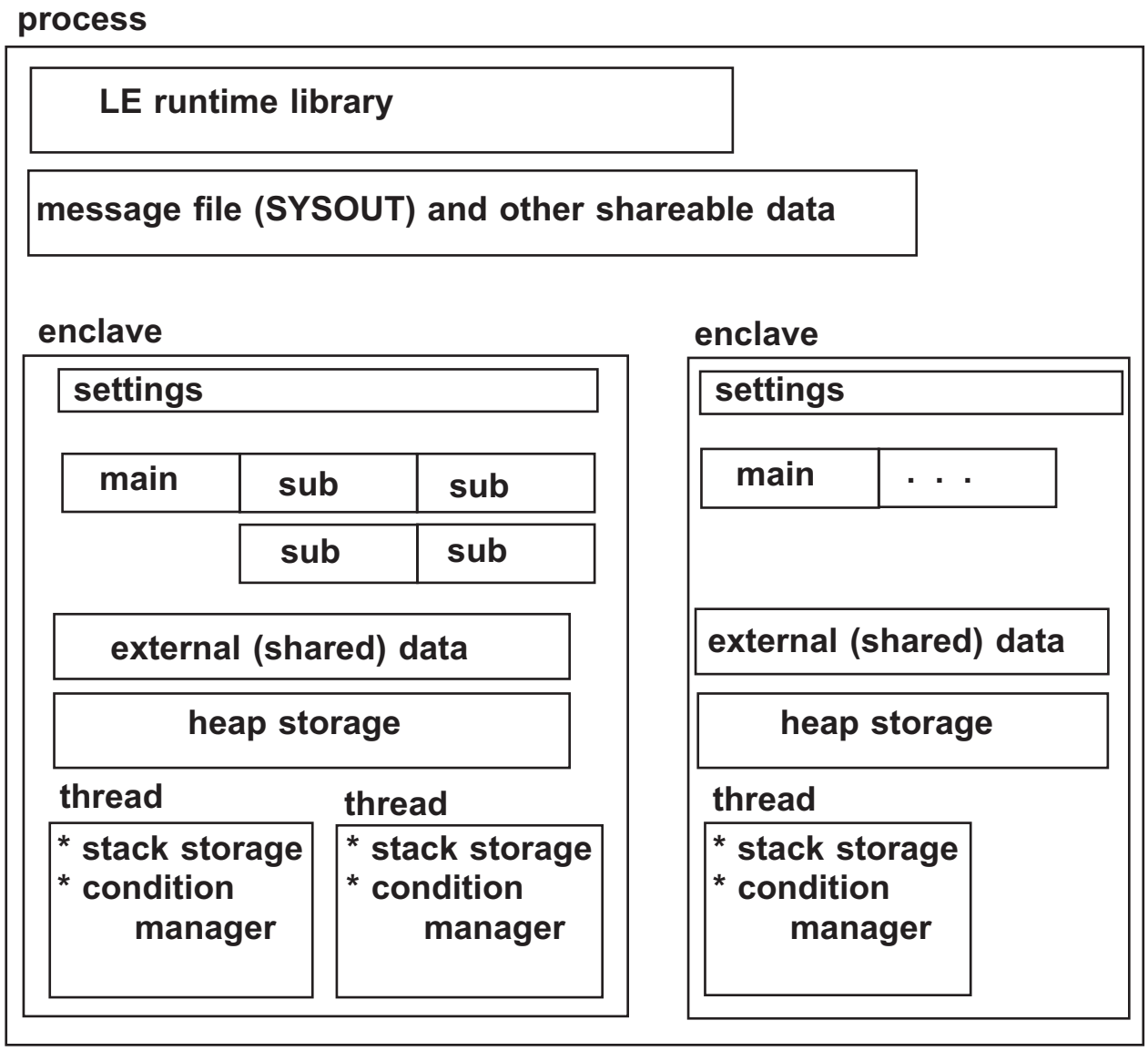
- ♦ **Consists of: one or more enclaves and process level resources**
  - ✗ There are currently no LE-supplied services for creating multiple enclaves in a process, but some CICS processes and some Assembler processes can create multiple enclaves in a process using non-LE services
  - ✗ A process can create other processes, although processes are independent of one another (no hierarchical relationships)

❒ **The resources managed at the process level, include**

- ♦ **Message file**
- ♦ **The Language Environment run-time library**

LE Program Management, concluded

❒ Diagramatically, here's how the pieces fit together in the LE program management model:



❒ Notes

- ♦ This represents the full model, which is not all implemented in the current version of z/OS
- ♦ We'll see single process, single enclave, single thread

Section Preview

- Debugging and Dump Reading
  - Onion (Machine Exercise)
  - Guidelines for Debugging
  - Sources of Information
  - Messages and Clues
  - Anatomy of a COBOL Compile Listing
  - Machine Instructions
  - Executable Programs
  - LE Dump Reading
  - Common Errors To Watch For
  - Lab Time for ONION

Computer Exercise: ONION

This is a special debugging exercise. Each individual or team will work with a copy of the program called ONION (real name: ONIONLCO).

To get going, you need to run a little dialog that will create files for you to use during the labs. From ISPF option 6, enter the following command:

```
===> ex   '________.train.library(d732strt)'   exec
```

This will prompt you for a high level qualifier to use for your libraries, set up with a default of your TSO id; if this is good (and it usually is), just press <Enter>. The dialog then will create three libraries:

<hlq>.TR.COBOL	- for your source code; contains ONIONLCO
<hlq>.TR.CNTL	- for your JCL; contains several members
<hlq>.TR.LOAD	- where programs are compiled into

Next, you need to run a couple of jobs from your TR.CNTL library, in preparation for our dump reading lectures. First submit member DUMPST3; this job compiles and binds a subroutine named XLINESE9; after this job completes, then submit member DUMPST4; this job compiles and binds the mainline named SUB3TST; this job also runs the resulting load module, which abends with a S0C7 code. We will be viewing both these jobs later, so save the jobnames and JOBIDs of these two jobs.

Now you are ready to get the debugging program, ONION, started...

Computer Exercise: ONION, continued

ONION is designed to blow up. Each time you get a dump, or other unusual termination, you are to use all your debugging skills to identify the precise cause of the failure and to suggest your approach to solve the problem.

Use member D732RUN1 in your TR.CNTL library to compile, bind, and run ONION. You can begin debugging any time you like.

Before submitting your proposed change(s) for another run, talk to the instructor. You must modify the current version just enough to correct the current error. This is because the program will reveal a new error after you fix the current one, until a total of ten or twelve errors have been corrected.

The current source code for ONIONLCO is found in the Appendix, along with the expected results, so you'll know when you're done.

An essential part of debugging is understanding what a program is designed to accomplish. On the next page is a brief description of ONION's functionality.

Computer Exercise: ONION, continued

Notes:

ONION reads an inventory file (INPUTA) and writes a report that lists each item. After reading all of the inventory file, ONION CALLs a module, INDXHD4 (the supplied JCL will automatically pick up this program at link time).

INDXHD4 was written by Peter Programmer, who is no longer with us. We can't seem to find the source of this program, and the only documentation we can locate is a cryptic note on the blotter Peter had on his desk: "INDXHD4: called passing request code ('T' for title line, 'D' for detail line), printarea, current table category, and current table category-count".

Anyway, INDXHD4 has never failed, so we're confident it is not the source of any errors.

The record layout for the input file is shown below:

INPUTA Record Layout

Positions	Data
1 - 9	Part number; character
10 - 39	Description; character
40 - 44	Reserved; random character string
45 - 48	Unit Price; packed decimal: 9999V999
49 - 51	Quantity on hand; packed decimal: 99999
52 - 52	Reserved
53 - 54	Quantity on order; binary halfword; 999
55 - 56	Reorder level (also used as reorder quantity); binary halfword; 999
57 - 57	Switch; random bit string
58 - 66	Old Part Number; character
67 - 67	Reserved
68 - 77	Item Category; character
78 - 100	Reserved