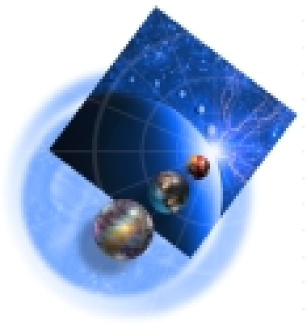
Assembler Issues When Migrating to LE and/or AMODE 31



Tom Ross
SHARE Session 8216
August, 2003

e-business

Introduction

- **Moving COBOL and/or PL/I applications with some assembler programs mixed in to LE**
- **There are cases when you can run your existing mixed assembler-COBOL and assembler-PL/I applications with LE without change!**
  - When can I do this?
  - When do I have to change my application?
  - What kinds of changes do I have to make?
- **There are some nice features in LE for assembler programs if the assembler programs are LE-conforming!**
- **How to upgrade assembler programs to AMODE 31**

e-business

Reference Material

- If you do not have one, get a COBOL Migration Guide! There is an appendix dedicated to the migration issues of mixed assembler/COBOL applications. Publication number: GC27-1409-01
- If you do not have one, get a PL/I Migration Guide! Publication number: SC26-3118
- How:
  - Call 800 879 2755 to order a hard copy
  - On the web at: www.ibm.com/s390/le/library/library.html
- See the LE Programming Guide for more information about running assembler with LE.

Terminology

- **Main program vs sub program**
  - In this presentation, the main program is the program that brings up LE.
  - A sub program is any program called from the main program or from another sub program while running under LE.
- **LE-conforming assembler**
  - An LE-conforming assembler program is a program that uses the LE provided macros to generate the required prolog and epilog code, and follows the register conventions of LE.
  - CEEENTRY and CEETERM
  - Details are in the LE Programming Guide.

LE-conforming assembler sample

- A simple LE-conforming assembler main program:

```
*=====================================================================
*    Bring up LE.
* =====================================================================
ASMLE3    CEEENTRY PPA=MAINPPA,AUTO=WORKSIZE,MAIN=YES
            USING WORKAREA,13
            LA     1,0                   Pass no parms
            L      15,A1C401B            Get the addr of the COBOL or PL/I pgm
            BALR   14,15                 Call it
* =====================================================================
*    Terminate LE.
* =====================================================================
            CEETERM   RC=0
MAINPPA   CEEPPA                         Constants describing the code block
A1C401B   DC     V(A1C401B)              VCON for COBOL or PL/I pgm
* ====================================================================
*          The Workarea and DSA
* ====================================================================
WORKAREA DSECT
            ORG     *+CEEDSASZ           Leave space for the DSA fixed part
            DS      0D
WORKSIZE EQU     *-WORKAREA
            CEEDSA                       Mapping of the Dynamic Save Area
            CEECAA                       Mapping of the Common Anchor Area
            CEEEDB                       Mapping of the Enclave Data Block
            END    ASMLE3
```

Can I run with LE without changes?

If your mixed assembler-HLL application meets the following criteria, there is a good chance it will run under LE without changes:

- **Your assembler programs follow the S/390 save area convention.**
  - R13 must contain the address of a save area.
  - The first two bytes of the save area must be hex zeros.
  - The back chain must be a valid 31-bit address. No garbage in the high order byte!
  - Each program needs its own save area (no sharing)
- **OS services ESPIE and ESTAE are not used.**
- **OS service DELETE is not used to delete load modules containing COBOL or PL/I programs.**
- **OS service LINK is not used to invoke OS/VS COBOL programs in more than one enclave.**

Can I run with LE without changes?

- **Some users thought they had to change their assembler programs to put the CAA address in R12**
  - This is not necessarily needed when running under LE!
  - Source of the problem was the LE Programming Guide in the section that talked about register conventions.
- **The book was updated for LE in OS/390 V2R6 to reflect the requirements for assembler and LE-conforming assembler.**
  - See the next page for the updated text.
  - When COBOL is called by assembler, LE always looks up the address of the CAA (so R12 does not need to point to the CAA).
  - When PL/I is called by assembler, LE does NOT look up the address of the CAA (so R12 DOES need to point to the CAA).
- **LE callable services expect R12 to point to the CAA.**

LE Programming Guide

5.2.2 Register Conventions

To communicate properly with assembler routines, you must observe certain register conventions on entry into the assembler routine (while it runs), and on exit from the assembler routine.

Language Environment-conforming assembler and non-Language Environment-conforming assembler each has its own requirements for register conventions when running under Language Environment.

LE Programming Guide

5.2.2 Language Environment-conforming Assembler

When you use the macros listed in "Assembler Macros" in topic 5.2.5 to write Language Environment-conforming assembler routines, the macros generate code that follows the required register conventions. On entry into the Language Environment-conforming assembler main routine, registers must contain the following values because they are passed without change to the CEEENTRY macro:

R0	Address of a parameter list, if the main routine is invoked from VM
R1	Address of the parameter list, or zero if no parameters are passed
R13	Caller's standard register save area
R14	Return address
R15	Entry point address

LE Programming Guide

5.2.2 Language Environment-conforming Assembler

On entry into a Language Environment-conforming assembler subroutine, these registers must contain the following values when NAB=YES is specified on the CEEENTRY macro:

R0	Reserved
R1	Address of the parameter list, or zero
R12	Common anchor area (CAA) address
R13	Caller's DSA
R14	Return address
R15	Entry point address

All others Undefined

LE Programming Guide

On entry into a Language Environment-conforming assembler routine, the caller's registers (R14 through R12) must be saved into the DSA provided by the caller. After you allocate a DSA (which sets the NAB field correctly in the new DSA), the first halfword of the DSA must be set to hex zero so the backchain will be set appropriately.

At all times while the Language Environment-conforming assembler routine is running, R13 must contain the executing routine's DSA.

At call points, R12 must contain the CAA address, except in the following cases:

* When calling a COBOL program
* When calling an assembler routine that is not Language Environment-conforming
* When calling a Language Environment-conforming assembler routine that specifies NAB=NO on the CEEENTRY macro

LE Programming Guide

On exit from a Language Environment-conforming assembler routine, these registers contain:

R0	Undefined
R1	Undefined
R14	Undefined
R15	Undefined
All others	The contents they had upon entry

5.2.2.2 Non-Language Environment-conforming Assembler

When you run a non-Language Environment-conforming routine in Language Environment, the following conventions must be followed:

* R13 must contain the executing routine's register save area
* The register save area back chain must be set to a valid 31-bit address (if the address is a 24 bit address, the first byte of the address must be hex zeros)

e-business

When changes are needed

- **There are cases when your assembler programs need to be changed. We will discuss the following areas:**
  - Save area conventions are not followed
  - OS LOAD/DELETE services are used
  - OS LINK with OS/VS COBOL
  - OS ATTACH
  - OS ESTAE/ESPIE
  - Altering the program mask

If save area conventions are not followed

- **Save area conventions must be followed:**
  - R13 must contain the address of a save area.
  - The first two bytes of the save area must be hex zeros.
  - The back chain must be a valid 31-bit address.
    - No garbage in the high order byte!
    - Already required under OS PL/I
    - The COBOL Migration Guide has more details on this.
  - The forward chain does not need to be set.
- **40XX abends occur when conventions not followed.**
  - ABEND U4083 abends occur when there are back chain problems. Back chain is used every time an assembler program calls a COBOL sub program.
  - Back chain is NOT used every time an assembler program calls a PL/I sub program. You either tell PL/I to initialize or you tell PL/I that the environment is already initialized.
    - Can be problems if using OPTIONS(COBOL) but calling from assembler.
    - LE does NOT initialize PL/I for OPTIONS(COBOL) like OS PL/I

OS LOAD and DELETE

- **What is allowed/supported in the context of sub programs (no restrictions for main programs):**
  - OS LOAD to load modules that contain only assembler and/or COBOL sub programs.
  - OS DELETE to delete load modules that contain:
    - OS/VS COBOL programs
    - COBOL programs compiled NORENT: VS COBOL II, COBOL for MVS & VM, or COBOL for OS/390 & VM
- **OS DELETE warning!!!**
  - Continuous OS LOAD/DELETE activity of modules containing COBOL programs compiled with RENT can cause accumulation of runtime control blocks.
  - If DELETE function is required, use another approach (we will discuss this later).

OS LOAD and DELETE

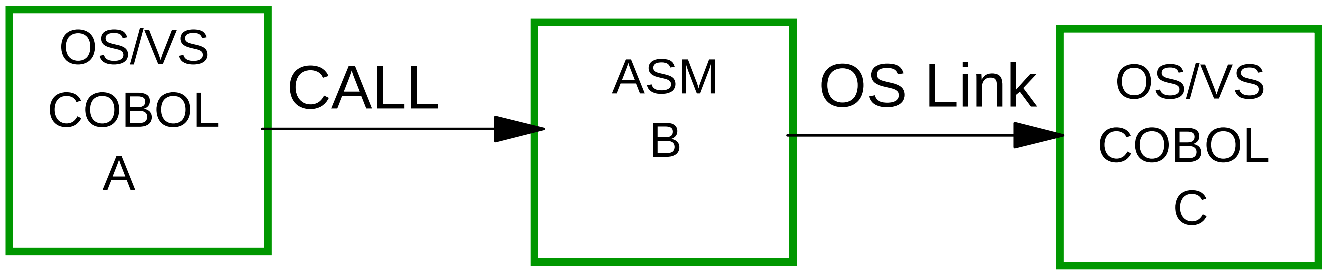
- **What won't work:**
  - OS LOAD and BALR to load modules that contain PL/I and the LE PL/I specific runtime is not initialized. See the PL/I migration guide.
  - OS DELETE to delete load modules that contain:
    - COBOL programs compiled RENT, any compiler
  - Use of OS DELETE can cause:
    - Storage accumulation of TGT, WORKING-STORAGE, plus other runtime control blocks
    - Program checks in the runtime

OS LOAD and DELETE

- **OS LOAD\DELETE alternatives**
  - Use a COBOL program to do the dynamic CALL and CANCEL processing.
  - Use a PL/I program to do the FETCH and RELEASE processing.
  - Use the LE assembler macros: CEEFETCH and CEERELES. Documentation is in the LE Programming Guide.
    - Requires that your assembler program be LE-conforming.
  - Use the vendor interface IGZCXCC (COBOL dynamic call/cancel from assembler). Documentation is in the LE Vendor Interfaces Book (SY28-1152).
    - Requires that your assembler program be LE-conforming.
    - Avoid using function codes 3 and 4 (call/cancel with entry point provided). These function codes are removed in OS/390 V2R6.

OS LINK with OS/VS COBOL

- **OS LINK with OS/VS COBOL**
  - OS LINK is allowed as long as OS/VS COBOL is run in only one LE enclave.
  - For example the following is not supported (it is diagnosed by LE with message IGZ0168S):



- **Alternatives:**
  - Recompile the COBOL programs with VS COBOL II or later
    - Must be done if OS LINK is retained for 'abend protection'
  - Use COBOL dynamic CALL (and CANCEL)
    - Instead of OS LINK, no 'abend protection' for caller

OS ATTACH

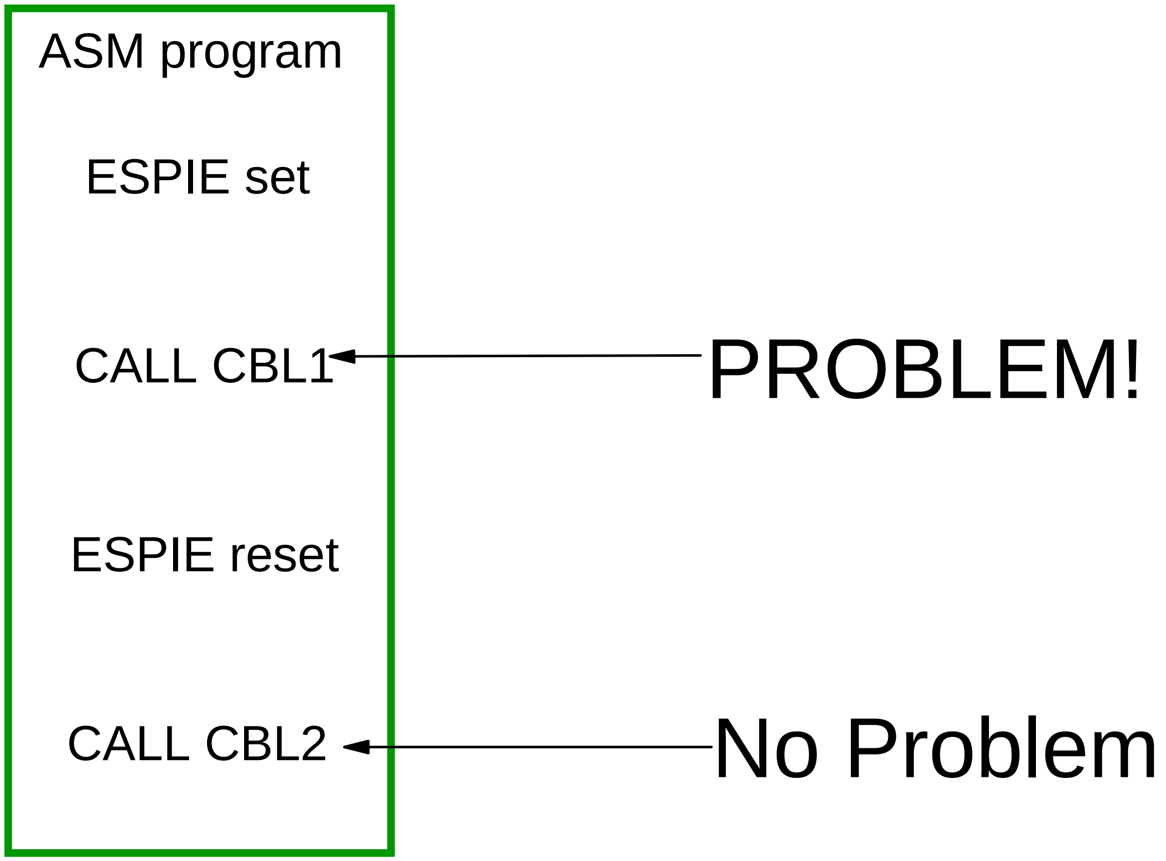
- **The default parameter list processing for main COBOL programs is different compared to the VS COBOL II runtime.**
  - LE always assumes a "PARM=" style of parameter list
  - VS COBOL II had an algorithm to decide if the incoming parameter should be processed as a "PARM=" style of parameter list or PLIST=OS style (the algorithm is documented in the COBOL Migration Guide).
- **Alternatives:**
  - Change the main program to LE-conforming assembler, and use PLIST=OS keyword in the CEEENTRY macro. Then have the assembler program ATTACH the COBOL program.
  - Modify the parameter list processing via the LE exit IGZEPSX to be compatible with the VS COBOL II behavior.
- **Note: For PL/I, we recommend using PL/I tasking facility.**

OS ESTAE/ESPIE

- **On non-CICS, LE issues its own ESPIE and ESTAE with TRAP(ON). On non-CICS, LE issues its own ESTAE with TRAP(ON,NOSPIE).**
  - If an assembler program issues an ESPIE or ESTAE when LE is running, and it remains in effect after leaving the program, it can lead to ABENDs and unexpected results.
    - There is no problem if the ESPIE is reset or ESTAE is cancelled before leaving the program or calling LE services
  - LE expects to get control for ABENDs and program interrupts.
  - LE is designed to recover from program interrupts and ABENDs.
    - The LE math library is designed to recover from program interrupts (designed for performance).
    - Condition handling is designed to recover from program interrupts and abends (PL/I ON UNITs, C SIGABND, etc)

OS ESTAE/ESPIE

- There is no problem if the ESPIE is reset or ESTAE is cancelled before leaving the program or calling LE services



OS ESTAE/ESPIE

- **OS ESTAE/ESPIE alternative 1**
  - Change your application to use LE condition handlers and no longer issue ESPIE or ESTAE.
- **For example, instead of calling an assembler program to set an ESPIE to catch data exceptions, you would make a call to register an LE condition handler.**
  - The LE condition handler would take the place of your ESPIE exit code.
  - All COBOL programs that are involved with the condition handling must be compiled with COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL.
- **Example code is in the back of this presentation. Also see the information in the COBOL Migration Guide.**

OS ESTAE/ESPIE

- **OS ESTAE/ESPIE alternative 2**
- **Change your application to use the LE compiler writer interfaces (CWI) in support of condition management.**
- **Callable services that are available include:**
  - CEE3ERP -- Support for User-Provided Error Recovery
  - CEE3SMS -- Set Machine State CWI
  - CEEMRCM -- Move resume cursor using a machine state
- **Documentation of the CWIs are in the LE Vendor Interfaces Book (SY28-1152).**
- **Assembler programs using the CWIs must be LE-conforming.**
- **Also see the text in APAR PQ14362 about coordinating ESPIEs/ESTAEs with LE.**

Setting the Program Mask

- **If you have assembler sub programs that alter the program mask, they must restore the program mask before returning to COBOL or PL/I**
  - Failure to restore the program mask could result in undetected data errors.
  - Issuing an ESPIE can alter the program mask

LE-conforming assembler

- **Here are some of the things you can do with LE-conforming assembler:**
  - Use any of the LE callable services (documented in the LE Programming Reference)
  - Date/time, math, storage management, program management, condition management, and more!
  - Use CEEFETCH and CEERELES for program management
  - Use the LE services in the LE Vendor Interface Book
    - Lower level services for those that really need it
  - Assembler programs that are main (non-CICS only)
  - Call COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL programs when running on CICS and non-CICS
  - Call PL/I for MVS & VM programs when running on CICS and non-CICS
    - Note: PL/I has some FETCH restrictions when called from programs that are LOADed. See APAR PQ13009.
  - CALL C functions

LE-conforming assembler update

- **News Flash! APAR PQ46427**
  - Support has been added for specifying more than one base register for the BASE parameter of the CEEENTRY macro.
  - When more than one register is specified, the registers must be enclosed in parentheses and separated by commas like this: BASE=(9,10,11)
  - Replace description of the BASE parameter with:

    BASE=
    Establishes the registers that you specify here as the base registers for this module. If you do not specify a value, register 11 is assumed; register 12 cannot be used. Also if you specify MAIN=YES, you cannot specify register 2 as a base register for the module. When more than one register is specified, the registers must be separated by commas and enclosed in parentheses. The same register cannot be specified more than once.
- **PQ46427 applies to LE V2R8, V2R9, V2R10, z/OS R2**
  - Included in base for z/OS R3 and later

Converting assembler to AMODE 31

- **More than just link-editing with AMODE=31!**
- **Sometimes just changing AMODE 24 to AMODE 31 in source is all you need to do**
- **Before you do that, check out the following first**
  - Use of the SPM (set program mask) instruction
  - LA instruction used to clear the high-order byte of a register
  - Address fields that are less than 4 bytes
  - Use of the ICM (insert characters under mask) instruction
  - AMODE of subprograms
  - AMODE of CALLing programs

Converting assembler to AMODE 31

- **For a good discussion of 31-bit addressability issues, see the OS/390 MVS Assembler Services Guide, Chapter 5, "Understanding 31-Bit Addressing."**
  - Chapter 5.2 has notes on converting from 24- to 31-bit addressing mode.
- **For 31-bit I/O issues, see DFSMS Macro Instructions for Data Sets,**
  - Chapter 2, "Non-VSAM Macros" contains information for each macro, regarding 31-bit addressability considerations.
  - Appendix A in Appendix 1.1 contains a table, indicating for each macro, whether it can be issued in 31-bit amode.
- **http://publibz.boulder.ibm.com:80/cgi-bin/ bookmgr_OS390/BOOKS/iea1a630/5.2.1**

Converting assembler to AMODE 31

- **Use of the SPM (set program mask) instruction**
  - Does the module depend on the instruction length code, condition code, or program mask placed in the high order byte of the return address register by a 24-bit mode BAL or BALR instruction?
    - One way to determine some of the dependencies is by checking all uses of the SPM (set program mask) instruction. SPM might indicate places where BAL or BALR were used to save the old program mask, which SPM might then have reset. The IPM (insert program mask) instruction can be used to save the condition code and the program mask.
- **LA instruction used to clear high-order byte of register**
  - This practice will not clear the high-order byte in 31-bit addressing mode
  - Use SR and ICM

Converting assembler to AMODE 31

- **Address fields that are less than 4 bytes**
  - Are any address fields that are less than 4 bytes still appropriate?
  - Make sure that a load instruction does not pick up a 4-byte field containing a 3-byte address with extraneous data in the high-order byte
  - Make sure that bits 1-7 are zero
- **Use of ICM (insert characters under mask) instruction**
  - The use of this instruction is sometimes a problem because it can put data into the high-order byte of a register containing an address, or
  - It can put a 3-byte address into a register without first zeroing the register.
  - If the register is then used as a base, index, or branch address register in 31-bit addressing mode, it might not indicate the proper address

Converting assembler to AMODE 31

- **AMODE of subprograms**
  - If AMODE 24 then shared data must be below 16M line
- **AMODE of CALLing programs**
  - If AMODE 24, might have to handle mode switching
  - COBOL dynamic CALL handles this for you

Other interesting items

- **OS ATTACH support for COBOL**
  - You can run COBOL programs with LE in multiple tasks when all of the COBOL programs are compiled with COBOL for MVS & VM, COBOL for OS/390 & VM or Enterprise COBOL for z/OS and OS/390
  - A separate LE is set up for each task
  - ISPF split screen now allows COBOL to be run from both screens without having to use NORES
- **Tips:**
  - Specify a separate LE MSGFILE per task
  - See the LE Programming Guide for more considerations
  - Note: For PL/I, we recommend using PL/I tasking facility.

Other interesting items

- **Performance Notes**
  - Assembler calling a COBOL sub program running on LE is faster than it is when running with the VS COBOL II runtime
  - Additional performance benefits when the called program is COBOL for MVS & VM, COBOL for OS/390 & VM or Enterprise COBOL (compared to VS COBOL II programs)
  - Call overhead performance measured in CPU time improved up to 8x when running minimum programs
  - A COBOL CALL to another COBOL program is always faster than a call from assembler to COBOL
  - A PL/I CALL to another PL/I program is about the same cost as a call from assembler to PL/I

Other interesting items

- **LE Storage Allocation on non-CICS**
  - OS/VS COBOL and VS COBOL II runtime allocated storage from subpool 0
  - LE allocates storage from subpool 1 and subpool 2.
- **Subpool 1 storage is for runtime storage. For example: runtime control blocks, LIBSTACK, ANYHEAP, and BELOWHEAP.**
- **Subpool 2 storage is for user storage. For example: HEAP and STACK.**
- **COBOL WORKING STORAGE for RENT programs will be allocated from HEAP**
  - (which is allocated from subpool 2).
- **Some storage is still allocated out of subpool 0 when running OS/VS COBOL programs and VS COBOL II programs under LE**

Items not covered here...

- **Items not covered here but are discussed in the COBOL Migration Guide:**
  - List of the possible combinations of calls involving COBOL programs and assembler programs and whether the calls are supported or not
  - Change in behavior when the same COBOL program (compiled RENT) is dynamically called from a COBOL program and also called from an assembler program via LOAD and branch
    - The same copy of WORKING-STORAGE is used when running in the same enclave

Condition handling example

```
CBL APOST,NODYNAM,RENT
*****************************************************************
* Sample COBOL MVS & VM application that was converted
* from an OS/VS COBOL program that used a SPIE for
* error recovery.
*
* Routines used:
*    A1C3CHA1 - Main driver routine. It reads records from a
*                   file and processes each record. If a program
*                   check occurs while processing the record
*                   display the record and keep going.
*    A1C3CHAX - The condition handler.
******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. A1C3CHA1.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    SELECT F1 ASSIGN TO DD1
              FILE STATUS IS FILE-STATUS.
 DATA DIVISION.
 FILE SECTION.
    FD   F1   RECORDING MODE IS F
              BLOCK   0    RECORD   40
              LABEL RECORD STANDARD
     DATA   RECORD R1.

01   R1.
     02 R1-CTR    PIC 9(10).
     02 R1-INFO   PIC X(30).
```

Condition handling example

```
  WORKING-STORAGE SECTION.

  01   FILE-STATUS PIC 99.
       88 EOF VALUE 10.

 *-------------------------------------------
 * Data items needed for condition handling
 *-------------------------------------------
  01 PROCPTR USAGE IS PROCEDURE-POINTER.

  01 TOKEN    PIC X(4) VALUE SPACES.

  01   RECOVERY-AREA EXTERNAL.
       05 RECOVERY-POINT           POINTER.
       05 RECOVERY-IN-PROGRESS    PIC X(01).
*----------------------------------------------------------------
*    START THE PROGRAM...
*----------------------------------------------------------------
 PROCEDURE DIVISION.

      DISPLAY 'A1C3CHA1: ENTERING A1C3CHA1....   '
*----------------------------------------------------------------
*     Open the input file.
*----------------------------------------------------------------
      OPEN INPUT F1
      IF FILE-STATUS NOT = 0 THEN
        DISPLAY
          'A1C3CHA1: OPEN FAILURE ON FILE F1, FILE STATUS: '
           FILE-STATUS
        PERFORM UNEXPECTED-ERROR
      END-IF
```

Condition handling example

```
*----------------------------------------------------------------
*     Set up a condition handler.
*----------------------------------------------------------------
      SET PROCPTR TO ENTRY 'A1C3CHAX'
      CALL 'CEEHDLR' USING PROCPTR, TOKEN, OMITTED
*----------------------------------------------------------------
*     Set up the resume point where the condition handler
*     can resume to (which is the next COBOL statement below).
*----------------------------------------------------------------
      MOVE SPACES TO RECOVERY-IN-PROGRESS
      CALL 'CEE3SRP' USING RECOVERY-POINT, OMITTED
*----------------------------------------------------------------
*     We get here for one of two cases. Handle the flow based
*     on each case.
*     1) We just sucessfully called CEE3SRP. We want to
*         keep going in this case.
*     2) We got control here due to condition handling.
*         Go to the paragraph that handles errors.
*
*     NOTE: the SERVICE LABEL is required.
*----------------------------------------------------------------
      SERVICE LABEL
      IF RECOVERY-IN-PROGRESS = 'Y' THEN
        MOVE SPACES TO RECOVERY-IN-PROGRESS
        GO TO PROGRAM-CHECK-OCCURED
      END-IF
```

Condition handling example

```
*----------------------------------------------------------------
*     Read all of the records in the file and process them.
*     If there is a data exeception while processing a record,
*     display the record, and continue processing.
*----------------------------------------------------------------
 PROCESS-RECORDS.
      PERFORM PROCESS-DATA UNTIL EOF
*     Unregister the condition handler.

      CALL 'CEEHDLU' USING PROCPTR OMITTED
      CLOSE F1
      DISPLAY 'A1C3CHA1: EXITING SUCCESSFULLY.'
      STOP RUN
      .

*----------------------------------------------------------------
*     Read the record and process it
*----------------------------------------------------------------
 PROCESS-DATA.

      PERFORM READ-RCD
      IF NOT EOF THEN
*   The following Add may cause a data exception, if garbage input
        ADD 1 TO R1-CTR
*   Put out some messages if all went well.

        DISPLAY 'A1C3CHA1: PROCESSING COMPLETED SUCCESSFULLY.'
        DISPLAY '             RECORD "' R1 '"'
      END-IF
      .
```

Condition handling example

```
*-----------------------------------------------------------------
*     Read a record from file F1.
*-----------------------------------------------------------------
 READ-RCD.

      READ F1
      IF FILE-STATUS > 10 THEN
        DISPLAY 'A1C3CHA1: I/O ERROR ON FILE F1, FILE STATUS: '
                  FILE-STATUS
        PERFORM UNEXPECTED-ERROR
      .
*-----------------------------------------------------------------
*     Control comes to this paragraph if a program check occurs
*     while processing a record. Display the bogus record and
*     continue processing.
*-----------------------------------------------------------------
 PROGRAM-CHECK-OCCURED.

      DISPLAY 'A1C3CHA1: DATA EXCEPTION ENCOUNTERED.'
      DISPLAY '            PROCESSING WILL CONTINUE.'
      DISPLAY '            RECORD "' R1 '"'
      GO TO PROCESS-DATA
      .
*-----------------------------------------------------------------
*     Unexpected error!
*-----------------------------------------------------------------
 UNEXPECTED-ERROR.

      DISPLAY 'A1C3CHA1: EXITING WITH AN ERROR.'
      MOVE 16 TO RETURN-CODE
      STOP RUN
```

Condition handling example

```
*-----------------------------------------------------------------
*  The condition handler
*-----------------------------------------------------------------
 CBL APOST OPT(FULL) LIB
 IDENTIFICATION DIVISION.
 PROGRAM-ID. A1C3CHAX RECURSIVE.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01    RECOVERY-AREA EXTERNAL.
       05 RECOVERY-POINT              POINTER.
       05 RECOVERY-IN-PROGRESS       PIC X(01).

 LINKAGE SECTION.

 01    CURRENT-CONDITION.
       05    FILLER                  PIC X(8).
       COPY CEEIGZCT.
       05    FILLER                  PIC X(4).
 01    TOKEN                         PIC X(04).
 01    RESULT-CODE                   PIC S9(09) BINARY.
       88    RESUME                      VALUE +10.
       88    PERCOLATE                   VALUE +20.
       88    PERC-SF                     VALUE +21.
       88    PROMOTE                     VALUE +30.
       88    PROMOTE-SF                  VALUE +31.

 01    NEW-CONDITION                 PIC X(12).
```

Condition handling example

```
 PROCEDURE DIVISION   USING CURRENT-CONDITION,
                                 TOKEN,
                                 RESULT-CODE,
                                 NEW-CONDITION.

*-----------------------------------------------------------------
*     If we are here due to a data exception, set the resume
*     point and resume. Otherwise
*     percolate the error (we don't want to handle it).
*
*     We could add more code and verify that the name of
*     routine we took the error in by using service CEE3GRN.
*-----------------------------------------------------------------
      IF CEE347 THEN
        MOVE 'Y' TO RECOVERY-IN-PROGRESS
        CALL 'CEEMRCE' USING RECOVERY-POINT, OMITTED
        SET RESUME TO TRUE
      ELSE
        SET PERCOLATE TO TRUE
      END-IF
      GOBACK
```