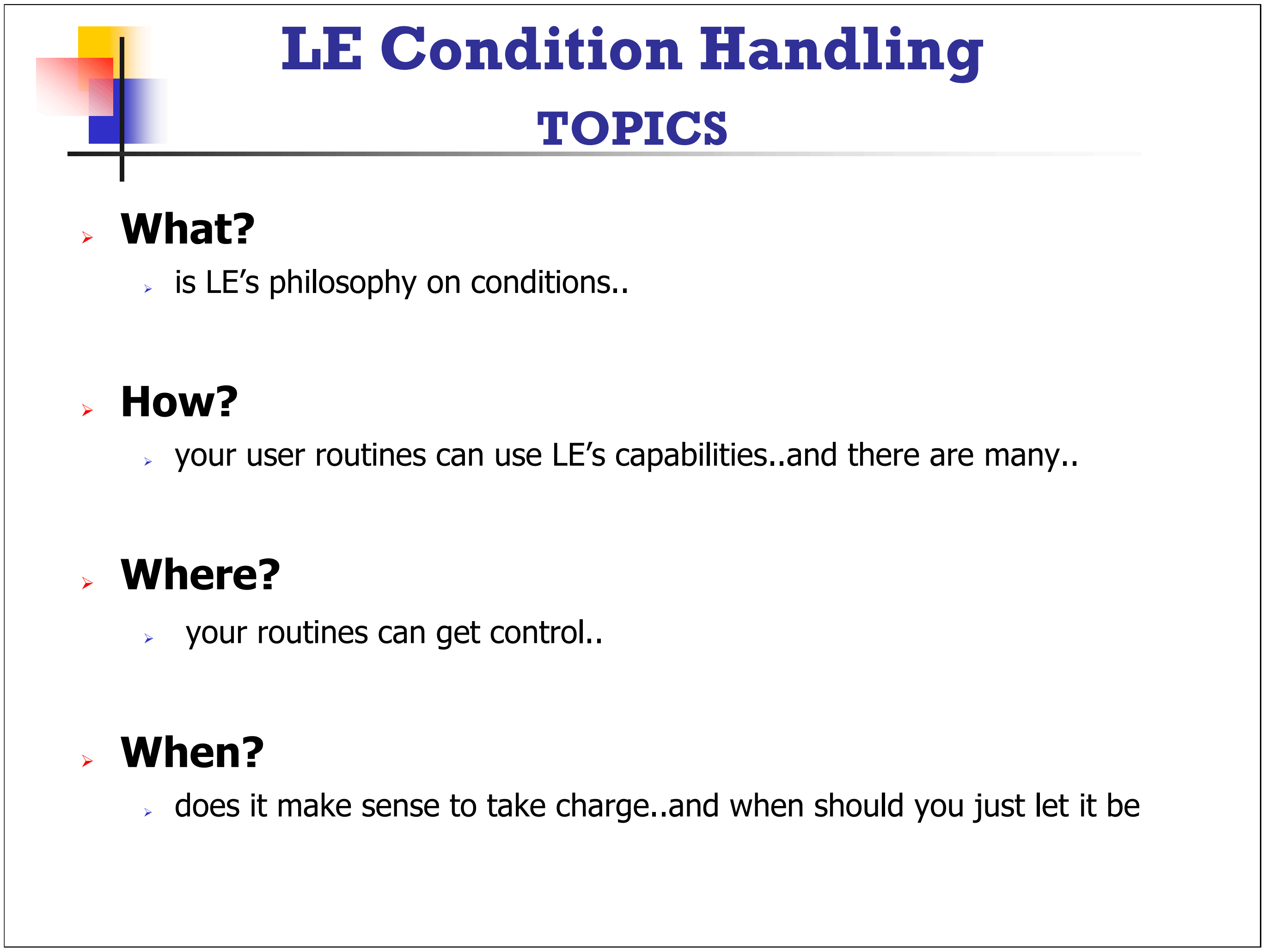
Condition Handling with Language Environment

Session #8234

Presented by: Janice Winchell
jwinche@us.ibm.com

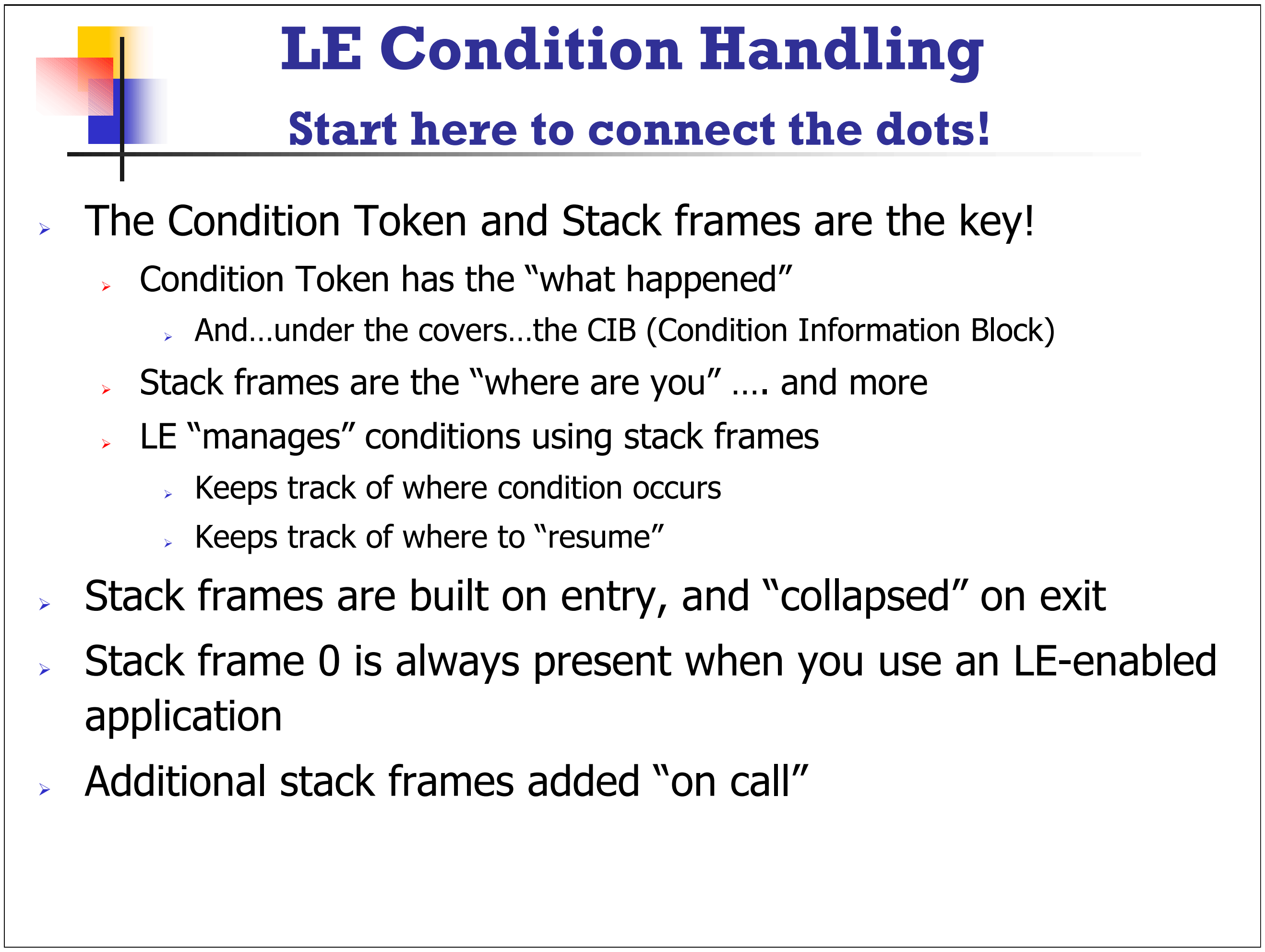
Trademarks and Copyright Notices



LE Condition Handling

TOPICS

- **What?**
  - is LE's philosophy on conditions..
- **How?**
  - your user routines can use LE's capabilities..and there are many..
- **Where?**
  - your routines can get control..
- **When?**
  - does it make sense to take charge..and when should you just let it be



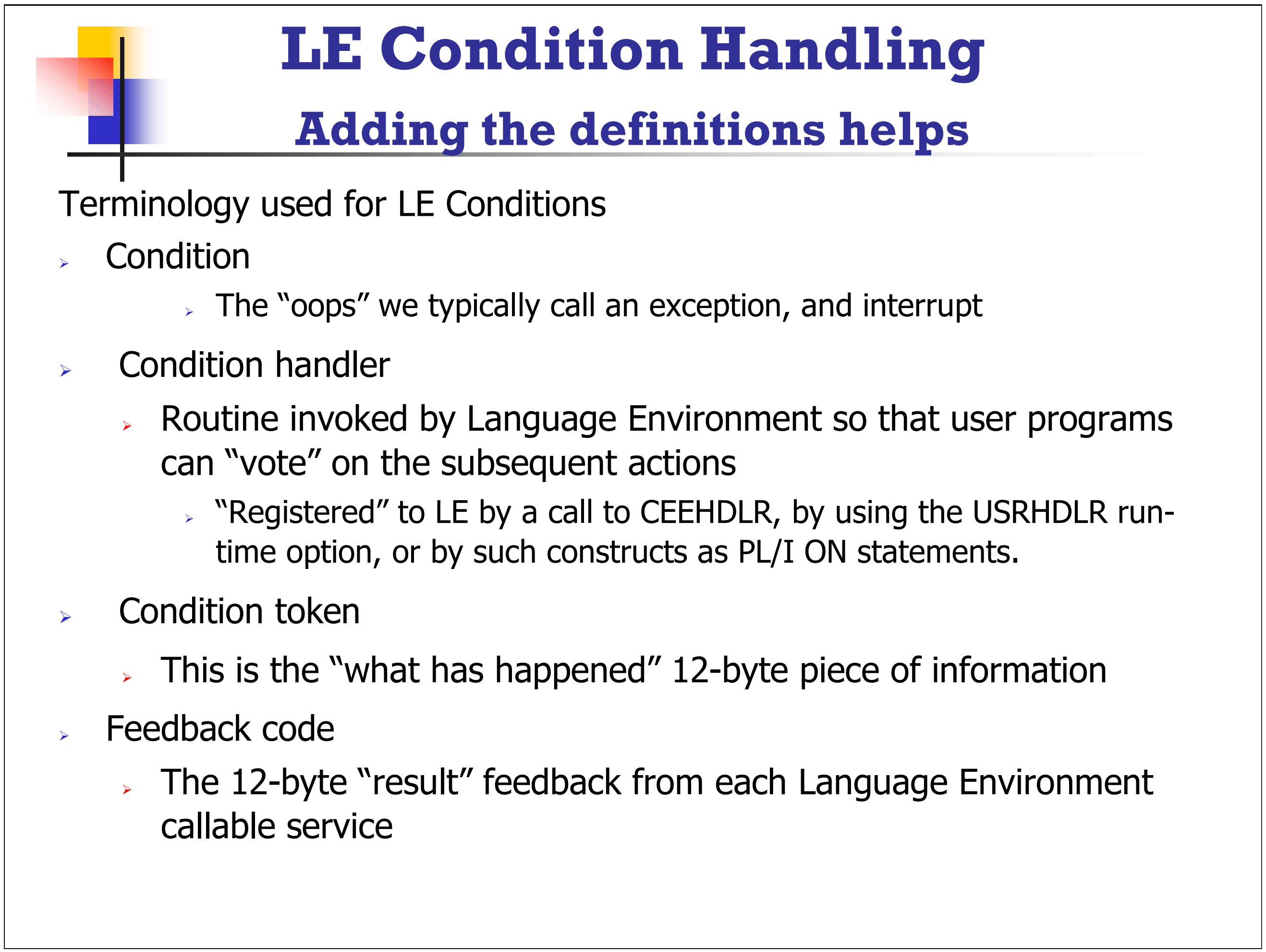
LE Condition Handling

Start here to connect the dots!

- The Condition Token and Stack frames are the key!
  - Condition Token has the "what happened"
    - And…under the covers…the CIB (Condition Information Block)
  - Stack frames are the "where are you" …. and more
  - LE "manages" conditions using stack frames
    - Keeps track of where condition occurs
    - Keeps track of where to "resume"
- Stack frames are built on entry, and "collapsed" on exit
- Stack frame 0 is always present when you use an LE-enabled application
- Additional stack frames added "on call"

Every "condition" will have a Condition Information Block (CIB). This is built by the condition manager and contains information built by the condition manager from LE. This block is not made available to user-written condition Handlers. It is not "intended to be viewed or used" by the user, but the information may be of interest!

The layout of the CIB can be found in the LE Debugging Guide and Run-time Messages manual. The key "pieces" of the condition token (it is a 12-byte area) for users are the *severity code*, the *message number*, and the *facility* issuing the message. This 12-byte code is the same layout as the feedback code returned from call's to CEExxx routines.



LE Condition Handling

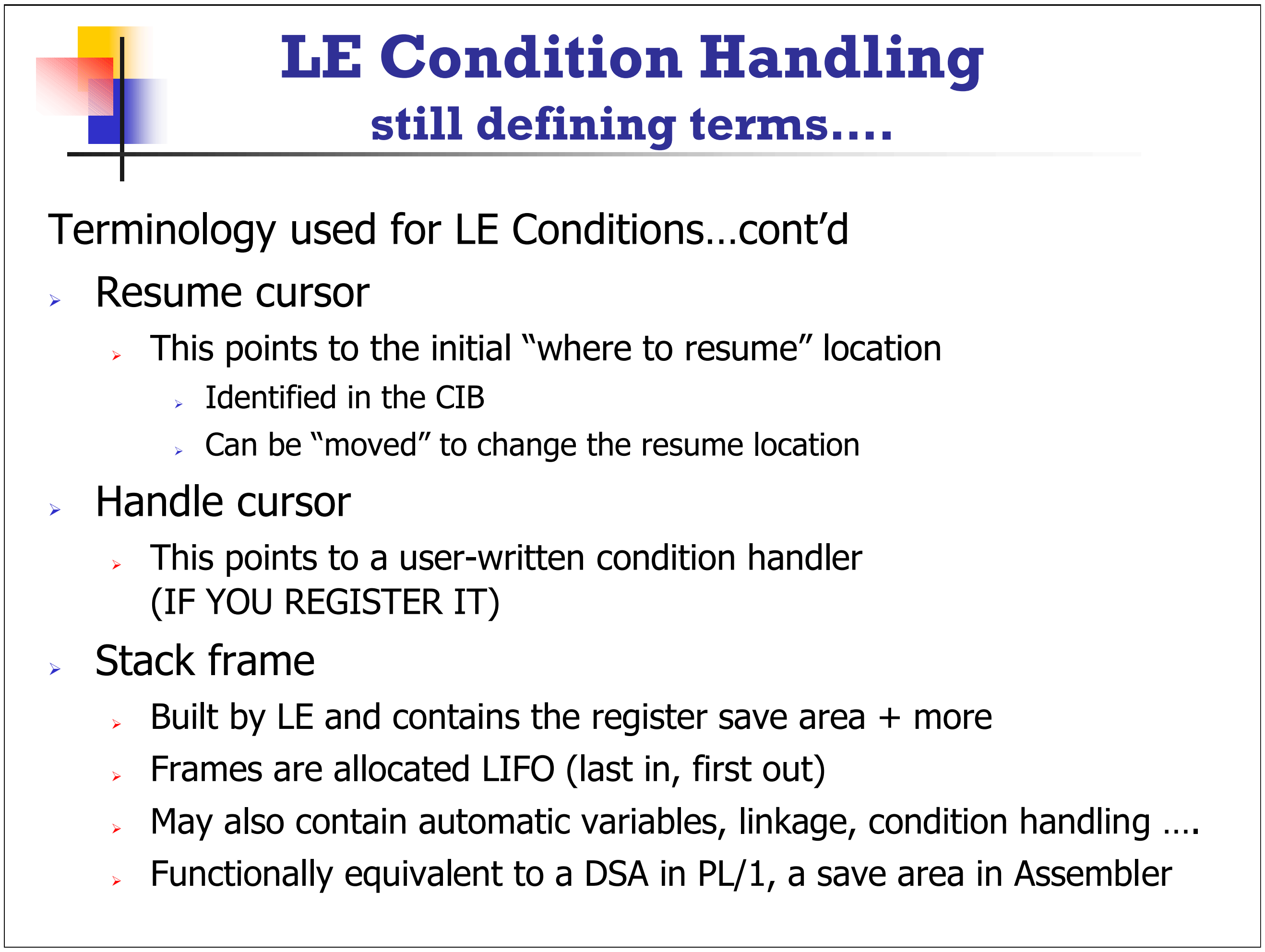
Adding the definitions helps

Terminology used for LE Conditions

- Condition
  - The "oops" we typically call an exception, and interrupt
- Condition handler
  - Routine invoked by Language Environment so that user programs can "vote" on the subsequent actions
    - "Registered" to LE by a call to CEEHDLR, by using the USRHDLR run-time option, or by such constructs as PL/I ON statements.
- Condition token
  - This is the "what has happened" 12-byte piece of information
- Feedback code
  - The 12-byte "result" feedback from each Language Environment callable service

The FEEDBACK code is considered "optional" when issuing calls to LE services. It contains extremely valuable information in case the call to the services "fail", so I would **strongly** recommend including it on each call to LE routines. Following the call you can check for a "zero" return – then everything is cool – but if not zero, then you can examine the FACILITY and MESSAGE to find out what went wrong. By combining the facility+message you get a message that you can look up in the LE Debugging Guide and Run-Time messages manual!

Of course, when using an LE facility, the "facility" 3-byte character code will be….you guessed it…'CEE'.



LE Condition Handling

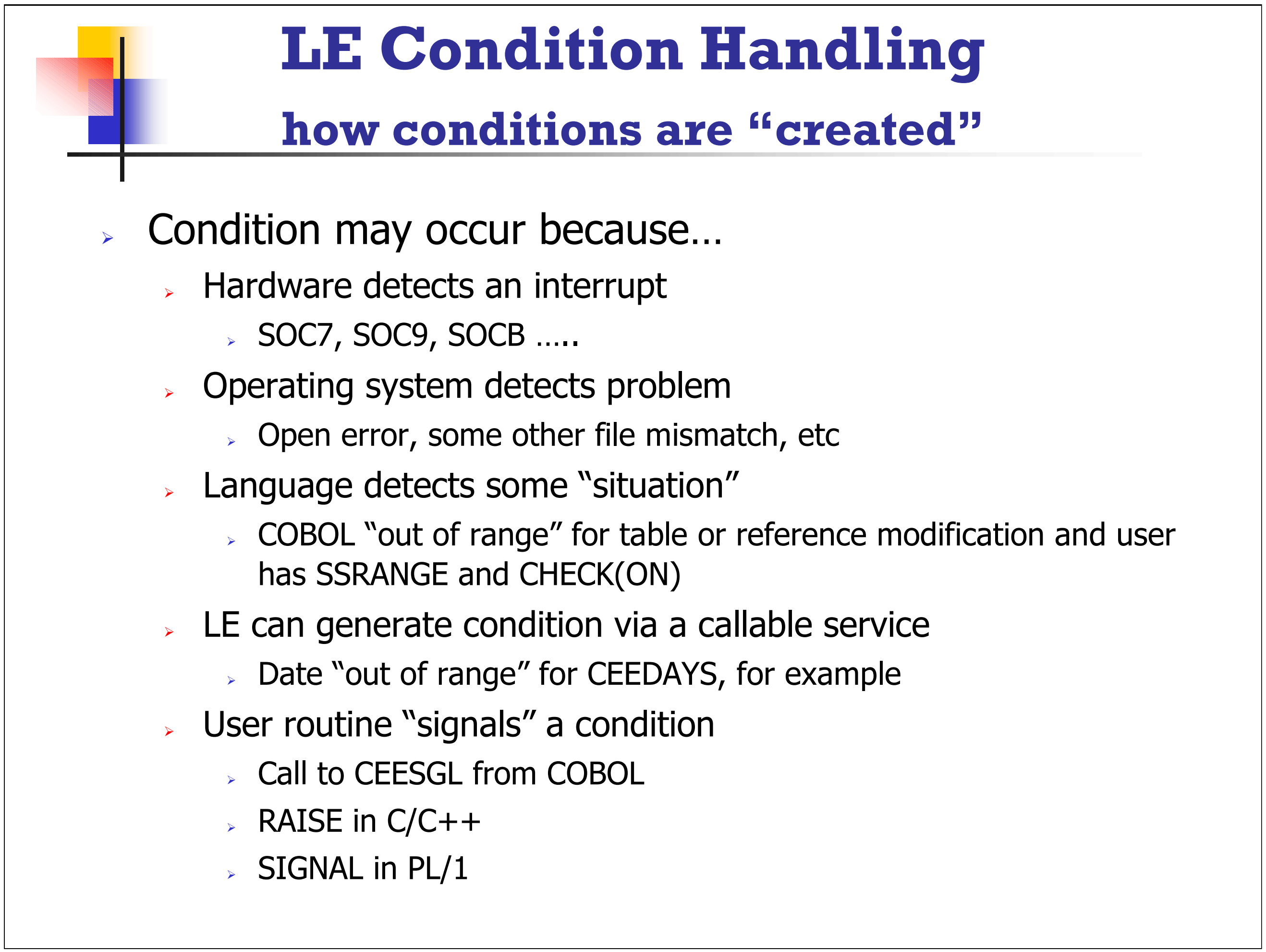
still defining terms….

Terminology used for LE Conditions…cont'd

- Resume cursor
  - This points to the initial "where to resume" location
    - Identified in the CIB
    - Can be "moved" to change the resume location
- Handle cursor
  - This points to a user-written condition handler (IF YOU REGISTER IT)
- Stack frame
  - Built by LE and contains the register save area + more
  - Frames are allocated LIFO (last in, first out)
  - May also contain automatic variables, linkage, condition handling ….
  - Functionally equivalent to a DSA in PL/1, a save area in Assembler

When using LE-Enabled assembler, CEEENTRY has the "prologue" code that constructs this area and does the register save, etc. CEETERM reloads registers and does the return. When you build LE-enabled Assembler it is no longer appropriate to have the traditional 18F savearea, and you no longer issue the SAVE and RETURN macros….this is all taken care of in the LE macros CEEENTRY CEETERM! For high-level languages, the save/return is also managed by Language Environment. The "prologue" of HLL's has a different look with Language Environment…no more 90ECD00C (STM 14,12,12(13)) at the beginning! In addition, the HLASM LE program is automatically constructed as AMODE(31), RMODE(ANY). The documented "solution" should you need to override this is to use the PARM='AMODE(31),RMODE(24)', for example, or get below the line storage and construct the DCB there, it just depends on the size of your handler. Since typically the handler is not EXTENSIVE code, it might be just as acceptable to use the PARM and ensure that the handler is loaded below the line. You typically would want the handler to be SEPARATE from the business executable code as a matter of good business practices, so overriding with the parm might just be the best practices solution!

Since you can use the SNAP (and other) macros that might need a DCB below the line, consider the "big picture" when coding user handlers in Assembler!



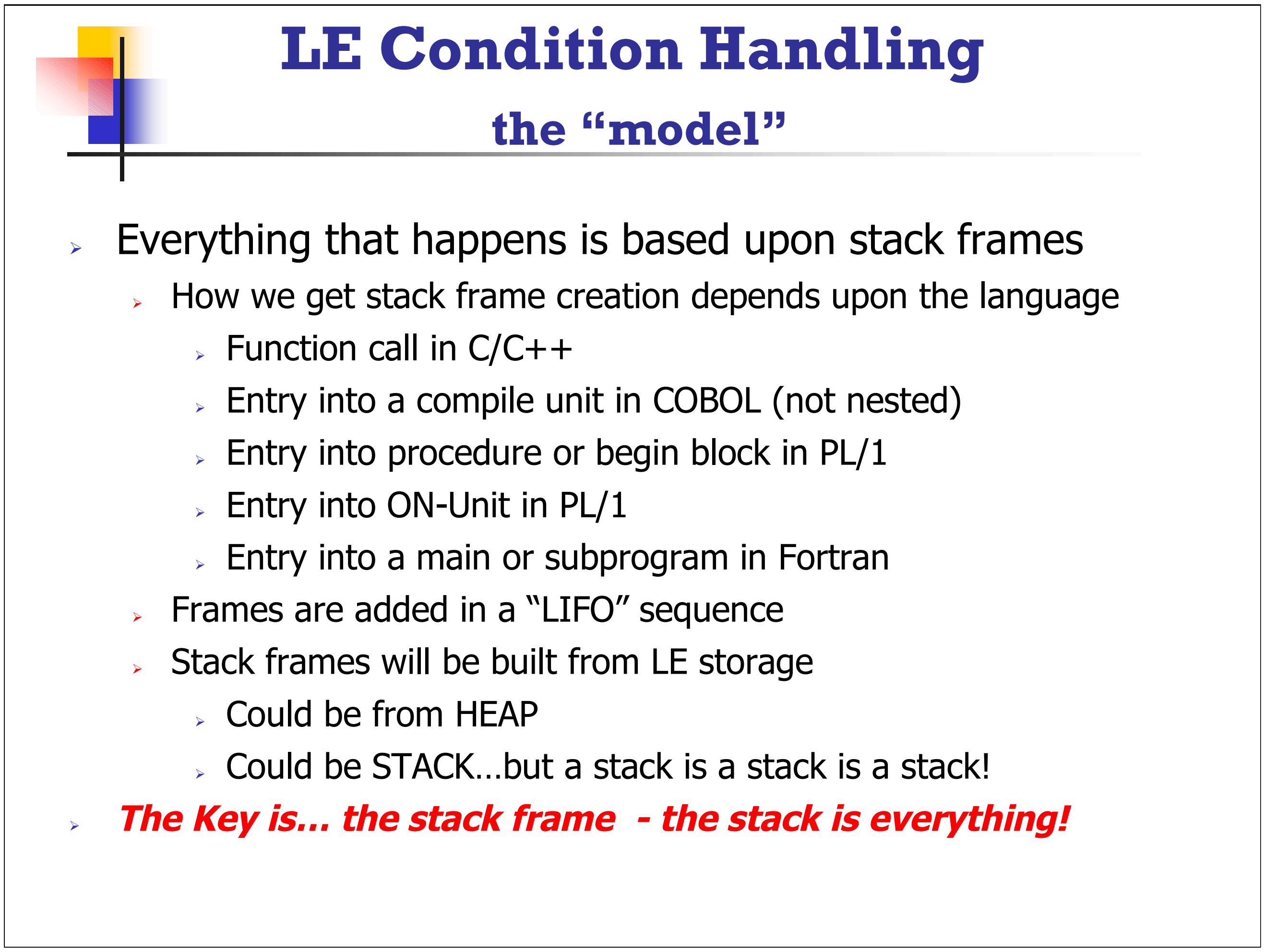
LE Condition Handling

how conditions are "created"

- Condition may occur because…
  - Hardware detects an interrupt
    - SOC7, SOC9, SOCB …..
  - Operating system detects problem
    - Open error, some other file mismatch, etc
  - Language detects some "situation"
    - COBOL "out of range" for table or reference modification and user has SSRANGE and CHECK(ON)
  - LE can generate condition via a callable service
    - Date "out of range" for CEEDAYS, for example
  - User routine "signals" a condition
    - Call to CEESGL from COBOL
    - RAISE in C/C++
    - SIGNAL in PL/1

When a condition is "signaled" LE does not care how the condition was set, LE will go through a normal sequence to check how to handle the condition. If your condition handler is a "regular" handler (not specified in a PARM as the super handler), normal processing sequence occurs.

If you code the PARM='/USRHDLR(NORMHDLR,SUPRHDLR)' then the SUPRHDLR gets control immediately! In the "normal" sequence, any HLL language semantic routines, such as ON SIZE ERROR would be used, if a size error occurred and ON SIZE ERROR was coded(COBOL), then user written handlers, then HLL-specific condition handlers, such as a PL/I ON-unit or a C signal handler, that might have been set up in the application.

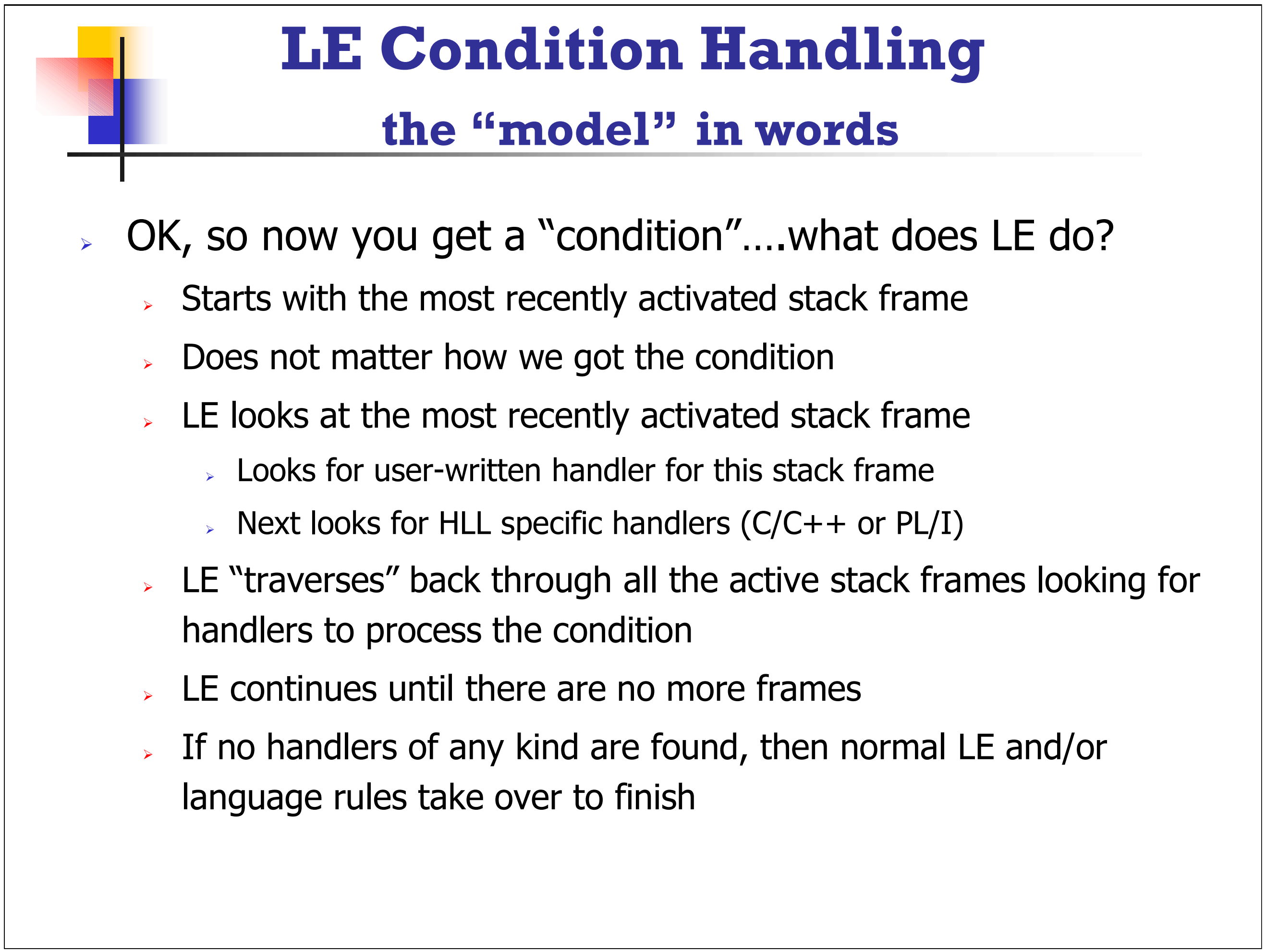


LE Condition Handling

the “model”

- Everything that happens is based upon stack frames
  - How we get stack frame creation depends upon the language
    - Function call in C/C++
    - Entry into a compile unit in COBOL (not nested)
    - Entry into procedure or begin block in PL/1
    - Entry into ON-Unit in PL/1
    - Entry into a main or subprogram in Fortran
  - Frames are added in a “LIFO” sequence
  - Stack frames will be built from LE storage
    - Could be from HEAP
    - Could be STACK…but a stack is a stack is a stack!
- ***The Key is… the stack frame - the stack is everything!***

LE acquires stack frame 0 to build and start, then successive frames are added depending on the language and the application.

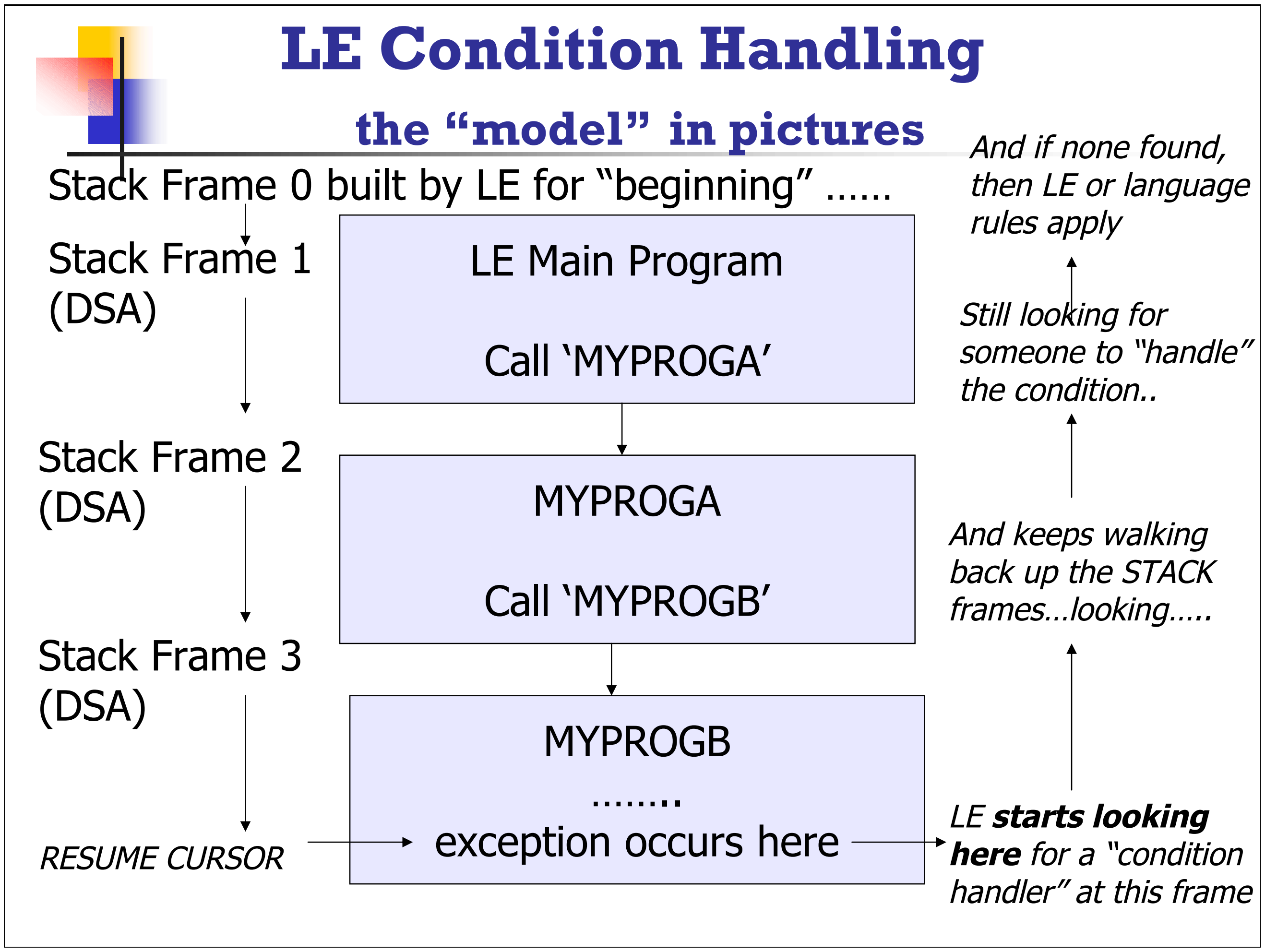


LE Condition Handling

the “model” in words

- OK, so now you get a “condition”….what does LE do?
  - Starts with the most recently activated stack frame
  - Does not matter how we got the condition
  - LE looks at the most recently activated stack frame
    - Looks for user-written handler for this stack frame
    - Next looks for HLL specific handlers (C/C++ or PL/I)
  - LE “traverses” back through all the active stack frames looking for handlers to process the condition
  - LE continues until there are no more frames
  - If no handlers of any kind are found, then normal LE and/or language rules take over to finish

LE will walk through the stack frames IN REVERSE ORDER, checking to see if there is a handler “registered”, that is, is there a HANDLE CURSOR at this frame? If there is then LE will load the handler program and execute it, if there are NO handler’s in effect, the normal language condition processing takes over.



LE Condition Handling

the "model" in pictures

Stack Frame 0 built by LE for "beginning" ……
Stack Frame 1 (DSA)
LE Main Program
Call 'MYPROGA'
Stack Frame 2 (DSA)
MYPROGA
Call 'MYPROGB'
Stack Frame 3 (DSA)
MYPROGB
……..
exception occurs here
RESUME CURSOR
LE **starts looking here** for a "condition handler" at this frame
And keeps walking back up the STACK frames…looking…..
Still looking for someone to "handle" the condition..
And if none found, then LE or language rules apply

Remember LE acquires a stack frame for

each CALL in COBOL

each FUNCTION in C/C++

each procedure or end block in PL/1

each ON-UNIT in PL/I.

Stack frame 0 is the starting point. The frames are actually built "up", not down as implied here, but it is easier to show the sequence of program to program, stack frame to stack frame!

LE Condition Handling

Resume Cursor

- Resume cursor points to the "where to resume" location
  - This cursor is always *on the move* as the programs execute, tracking the NSI (next sequential instruction)
  - Positioned after the instruction that causes the condition or signal
    - Next ***machine instruction***, not necessarily the next "source" instruction
  - May be "moved" with CEEMRCR LE callable routine to move relative to "here"
  - May be "moved" with CEEMRCE callable routine to move to an "explicit" location

Resume Cursor is just like the PSW "next" instruction address. Note than in high level languages this may be a generated instruction from the original coded statement, it is, truly, the next sequential instruction. If you do NOTHING, and say, respond with RESUME, then you effectively "step over" the offending instruction, but you may also execute some meaningless instructions generated by the high-level language, things like OI to "fix" the sign when no calculation actually occurred (say you failed on an packed decimal instruction).

You also have some added capability by MOVING the resume cursor and identifying SPECIFICALLY where you want to RESUME EXECUTION. Examples of these capabilities are in shown later in this presentation.

CEEMRCR - Moves resume cursor relative to the current positioned handle cursor. This can be similar to performing a GOTO, or setjmp() and longjmp(). The "move" depends upon the setting of the "type" to move the resume cursor to the **call return point** of the **routine registering the executing condition handler**, or to move the resume cursor to the **caller of the routine registering the executing condition handler.**

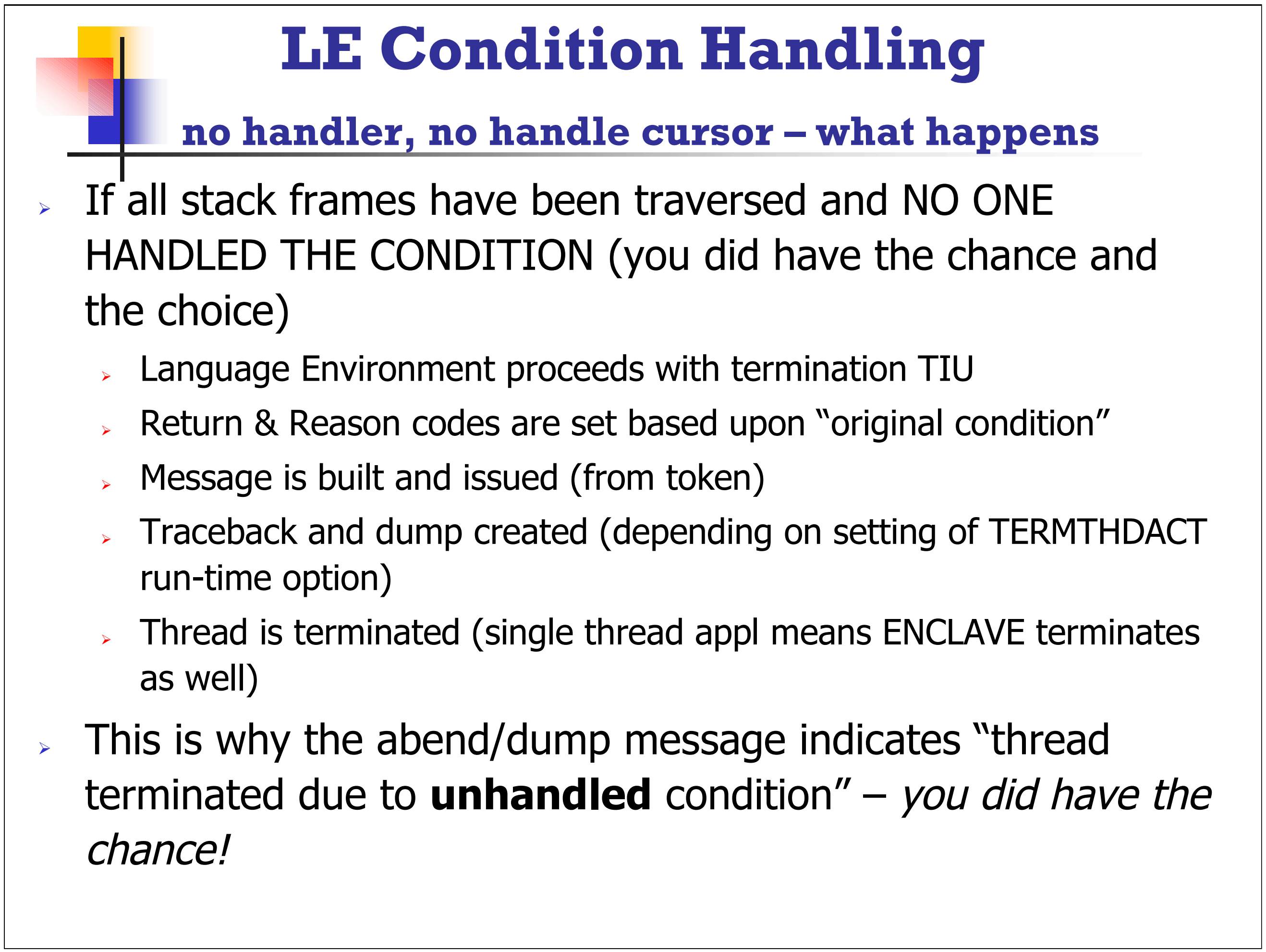
CEEMRCE - Moves the resume cursor to an explicit location where you want to resume after a condition has been handled. You must have previously set the "location" with a call to CEESRP - Set Resume Point

LE Condition Handling

Handle Cursor

- The other important element in “conditions” is the “handle cursor”
  - If you CALL CEEHDLR from your routine you have a handle cursor “registered” at this frame (where you issue this CEEHDLR call)
  - If you use the run-time option USRHDLR to identify a handler routine, you have a handle cursor “registered” at Stack Frame 0
  - This is the other piece of the LE CONDITION HANDLING “puzzle”
  - This is how LE knows whether to hand off the condition, and who to hand off to!

The “handle cursor” will exist only if you identify a user handler routine to Language Environment. You can use the “super handler” technique, with a run-time option to register a user handler IMMEDIATELY, or, you can use the Language Environment CEEHDLR routine to register a handler. At the point you “register” your handler, the handle cursor is in effect, and the identified handler programs will have a chance to look at abends that occur to “decide” whether to continue processing, RESUME, or not, PERCOLATE. You can have MULTIPLE handlers registered, and LE will proceed in LIFO order to let the handlers examine the condition. The super handler, registered via the run-time parm USRHDLR is often called the “super handler” as this handler routine gets invoked ahead of any other user registered condition handlers (via CEEHDLR registration).



LE Condition Handling

no handler, no handle cursor – what happens

- If all stack frames have been traversed and NO ONE HANDLED THE CONDITION (you did have the chance and the choice)
  - Language Environment proceeds with termination TIU
  - Return & Reason codes are set based upon "original condition"
  - Message is built and issued (from token)
  - Traceback and dump created (depending on setting of TERMTHDACT run-time option)
  - Thread is terminated (single thread appl means ENCLAVE terminates as well)
- This is why the abend/dump message indicates "thread terminated due to **unhandled** condition" – *you did have the chance!*

TIU – **T**ermination **I**mminent due to **U**nhandled condition

Termination activity can either be Language Environment or the High Level Language (HLL).

The **TERMTHDACT** options control the amount of information produced and may be:

TRACE – a traceback only

QUIET - no message

MSG - message only indicating what happened

DUMP - Message, Traceback, and CEEDUMP (no region or address space or program code)

UADUMP – Message, Traceback, CEEdump and U4039 which will create dump of address space

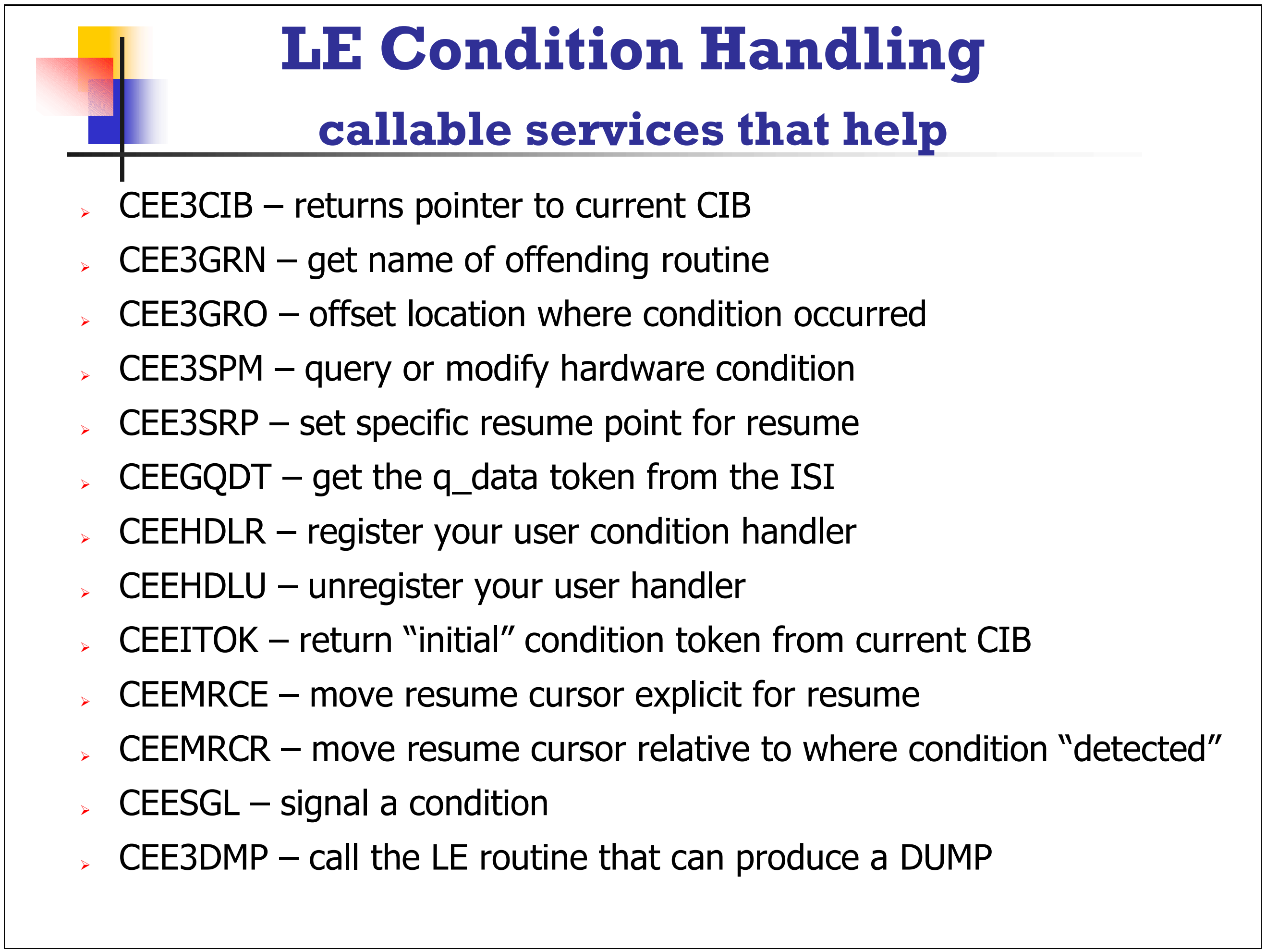
Note: You need a DD for SYSUDUMP or SYSMDUMP, for CICS you will get TRANSACTION DUMP

UATRACE – Message, Trace, and address space dump (no CEEDUMP)

UAONLY - LE will issue a U4039 abend which will drive an address space dump as long as you have the appropriate DD statements

UAIMM –Creates an address space dump before LE "handles" anything

Note: For software-raised conditions or signals, UAIMM behaves the same as UAONLY. When TRAP(ON,SPIE) is in effect, UAIMM will yield UAONLY behavior.



LE Condition Handling

callable services that help

- CEE3CIB – returns pointer to current CIB
- CEE3GRN – get name of offending routine
- CEE3GRO – offset location where condition occurred
- CEE3SPM – query or modify hardware condition
- CEE3SRP – set specific resume point for resume
- CEEGQDT – get the q_data token from the ISI
- CEEHDLR – register your user condition handler
- CEEHDLU – unregister your user handler
- CEEITOK – return "initial" condition token from current CIB
- CEEMRCE – move resume cursor explicit for resume
- CEEMRCR – move resume cursor relative to where condition "detected"
- CEESGL – signal a condition
- CEE3DMP – call the LE routine that can produce a DUMP

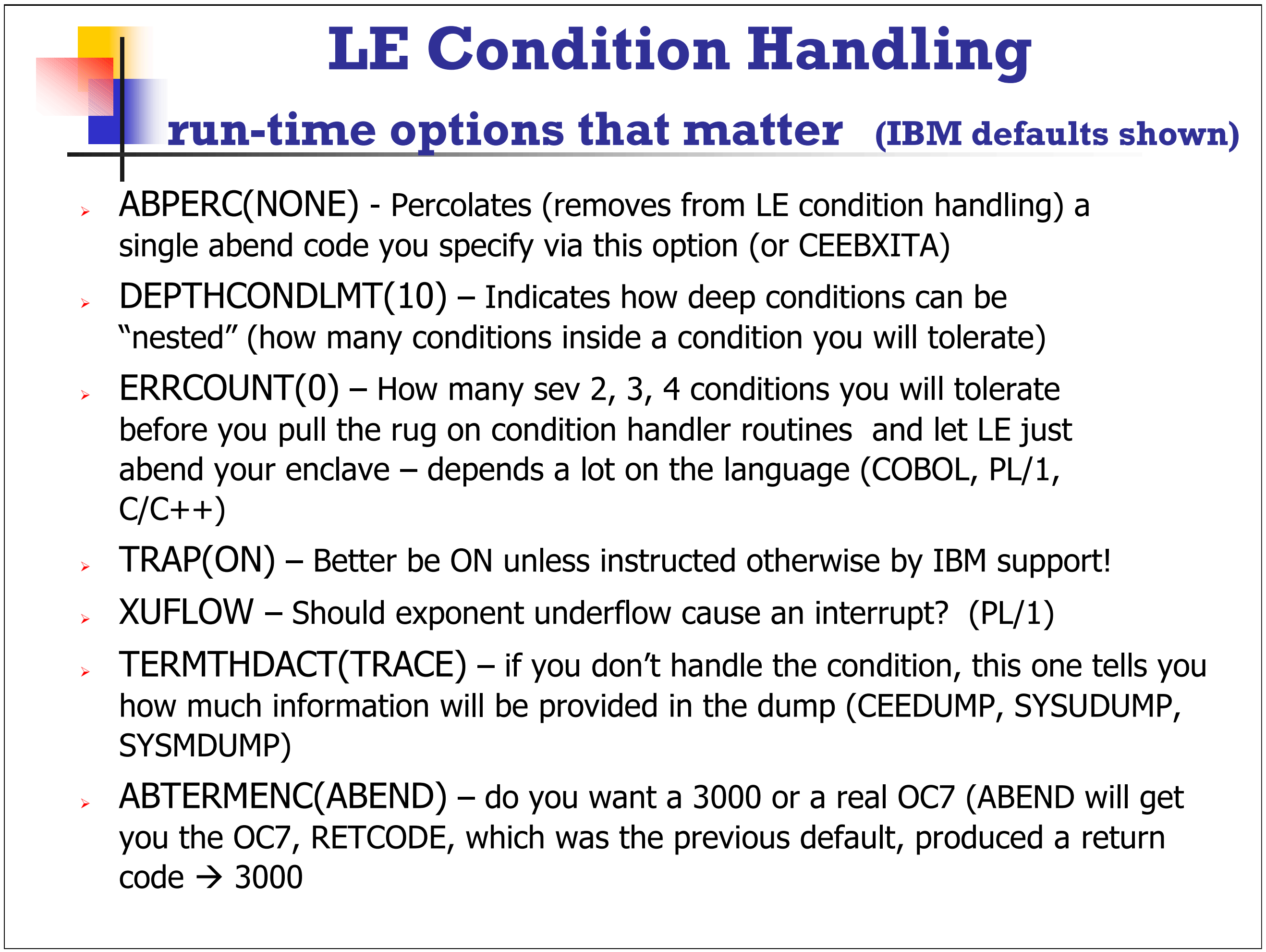
A CIB (Condition Information Block) created for each condition encountered. The address of the current CIB is kept in the CAA (Common Anchor Area).

For PL/1 and COBOL, the "map" to these fields is supplied in SCEESAMP.

For C/C++ the layout is in leawi.h.

This is basically the "fount of all knowledge" for conditions. Pointers and addresses in here may prove very valuable, should you need them! There is also a layout of the CIB in the LE Debugging Guide and Run-time messages.

There is an extensive descriptive discussion of all the fields in the CIB in the Language Environment for OS/390 & VM Vendor Interfaces (document number SY28-1152) which is part of the LANGUAGE ENVIRONMENT library.



LE Condition Handling

run-time options that matter (IBM defaults shown)

- ABPERC(NONE) - Percolates (removes from LE condition handling) a single abend code you specify via this option (or CEEBXITA)
- DEPTHCONDLMT(10) – Indicates how deep conditions can be "nested" (how many conditions inside a condition you will tolerate)
- ERRCOUNT(0) – How many sev 2, 3, 4 conditions you will tolerate before you pull the rug on condition handler routines and let LE just abend your enclave – depends a lot on the language (COBOL, PL/1, C/C++)
- TRAP(ON) – Better be ON unless instructed otherwise by IBM support!
- XUFLOW – Should exponent underflow cause an interrupt? (PL/1)
- TERMTHDACT(TRACE) – if you don't handle the condition, this one tells you how much information will be provided in the dump (CEEDUMP, SYSUDUMP, SYSMDUMP)
- ABTERMENC(ABEND) – do you want a 3000 or a real OC7 (ABEND will get you the OC7, RETCODE, which was the previous default, produced a return code → 3000

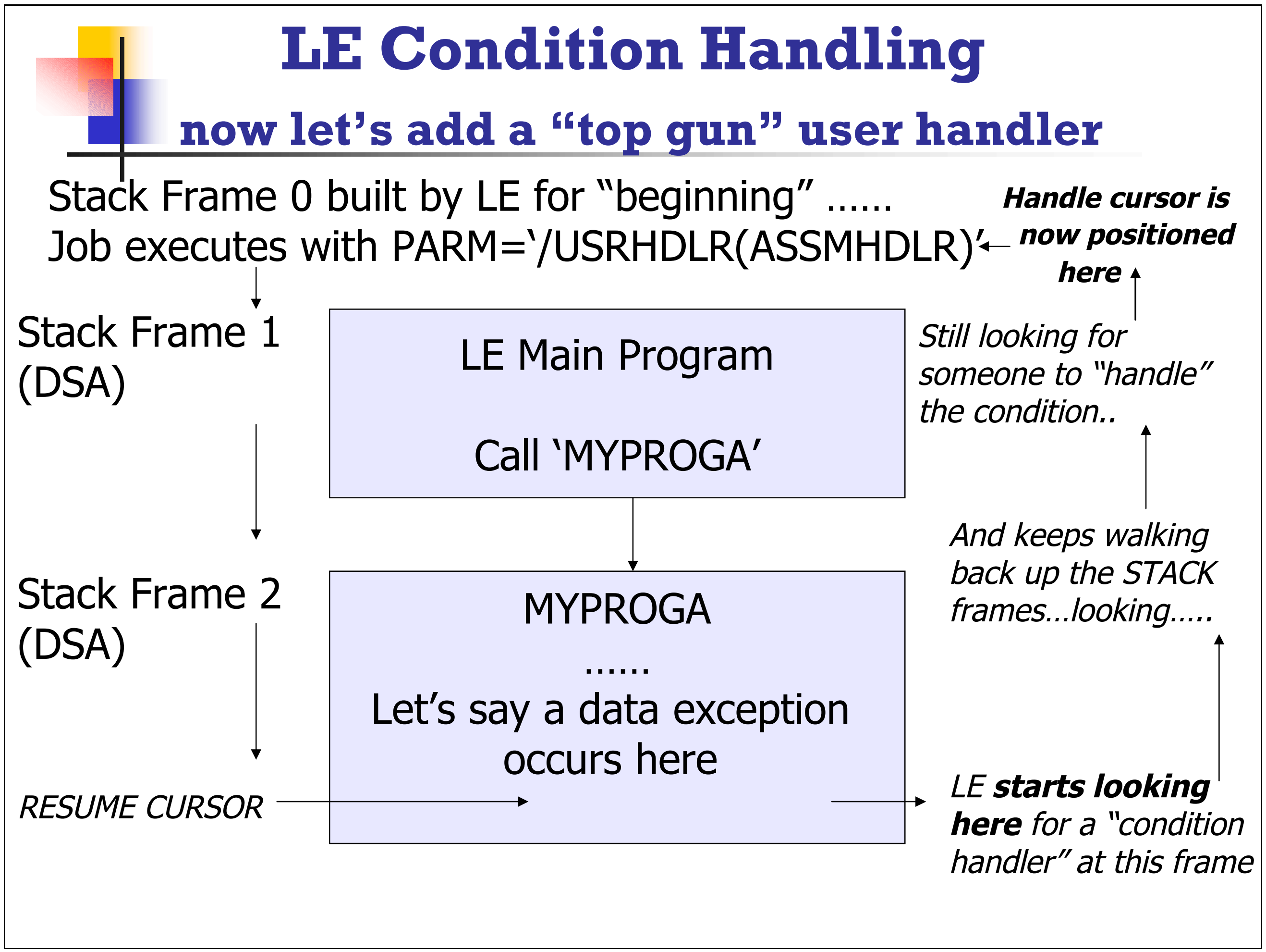
ERRCOUNT(0) means that the Language Environment condition handler will not terminate the task regardless of the severity 2, 3, or 4 conditions that are generated.

ERRCOUNT applies when conditions are handled by a user condition handler, signal catcher, PL/I on-units, or a language-specific condition handler. Language Environment does not count severity 0 or 1 messages. There are lots of language specific rules and information on ERRCOUNT in the following IBM manual:

OS/390 Language Environment for OS/390 & VM

Programming Reference Document Number SC28-1940-10

For COBOL it is possible to set ERRCOUNT(20) so that after 20 abnormal "conditions" have been handled, the next condition will terminate with no "handling". This gets a bit trickier with PL/1 and C/C++ since every SIGNAL and ON-units count toward ERRCOUNT.



LE Condition Handling

now let's add a "top gun" user handler

Stack Frame 0 built by LE for "beginning" ……
Job executes with PARM='/USRHDLR(ASSMHDLR)'

Handle cursor is now positioned here

Stack Frame 1 (DSA)

LE Main Program

Call 'MYPROGA'

Still looking for someone to "handle" the condition..

And keeps walking back up the STACK frames…looking…..

Stack Frame 2 (DSA)

MYPROGA

……

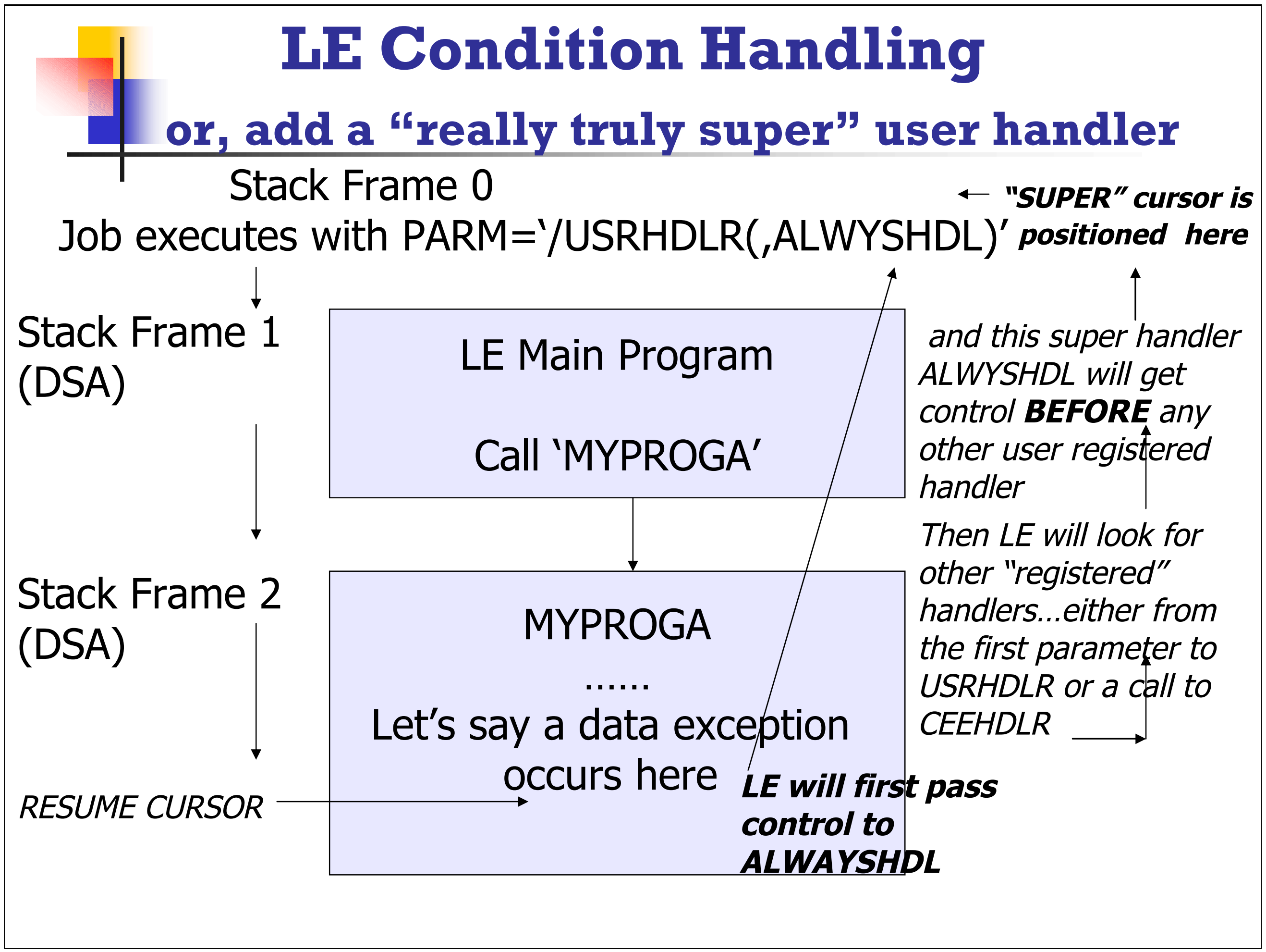
Let's say a data exception occurs here

RESUME CURSOR

LE ***starts looking here*** *for a "condition handler" at this frame*

NOTE: This PARM is a typical example for COBOL, where user parms precede LE run-time parameters separated by the slash. If you are using the standard LE parameter format, then the parm would be: PARM='USRHDLR(ASSMHDLR) where this is a standard user condition handler registered at run-time.

Handle Cursor at stack frame 0 has some differences from a routine registered via a call to CEEHDLR. There are some cross-communication opportunities, such as setting the RETURN-CODE, that cannot be done in COBOL programs with a "super" handler. C/C++ does recognize the return-code set via CEESRC called LE routine, as does PL/1, but COBOL uses COBOL's RETURN-CODE. What this means is, with a super handler and COBOL, you can intercept the abend, produce a dump using CEE3DMP, but you cannot indicate via a RETURN-CODE in your application that something "not good" has occurred. As you will see, with the facility of registering a handler using CEEHDLR which requires a bit of added code into your main COBOL program, you can produce the dump information AND CROSS-COMMUNICATE so you can indicate via a RETURN-CODE that something has occurred.



LE Condition Handling

or, add a "really truly super" user handler

Stack Frame 0

Job executes with PARM='/USRHDLR(,ALWYSHDL)'

"SUPER" cursor is positioned here

Stack Frame 1 (DSA)

LE Main Program

Call 'MYPROGA'

*and this super handler ALWYSHDL will get control **BEFORE** any other user registered handler*

Then LE will look for other "registered" handlers…either from the first parameter to USRHDLR or a call to CEEHDLR

Stack Frame 2 (DSA)

MYPROGA

……

Let's say a data exception occurs here

LE will first pass control to ALWAYSHDL

RESUME CURSOR

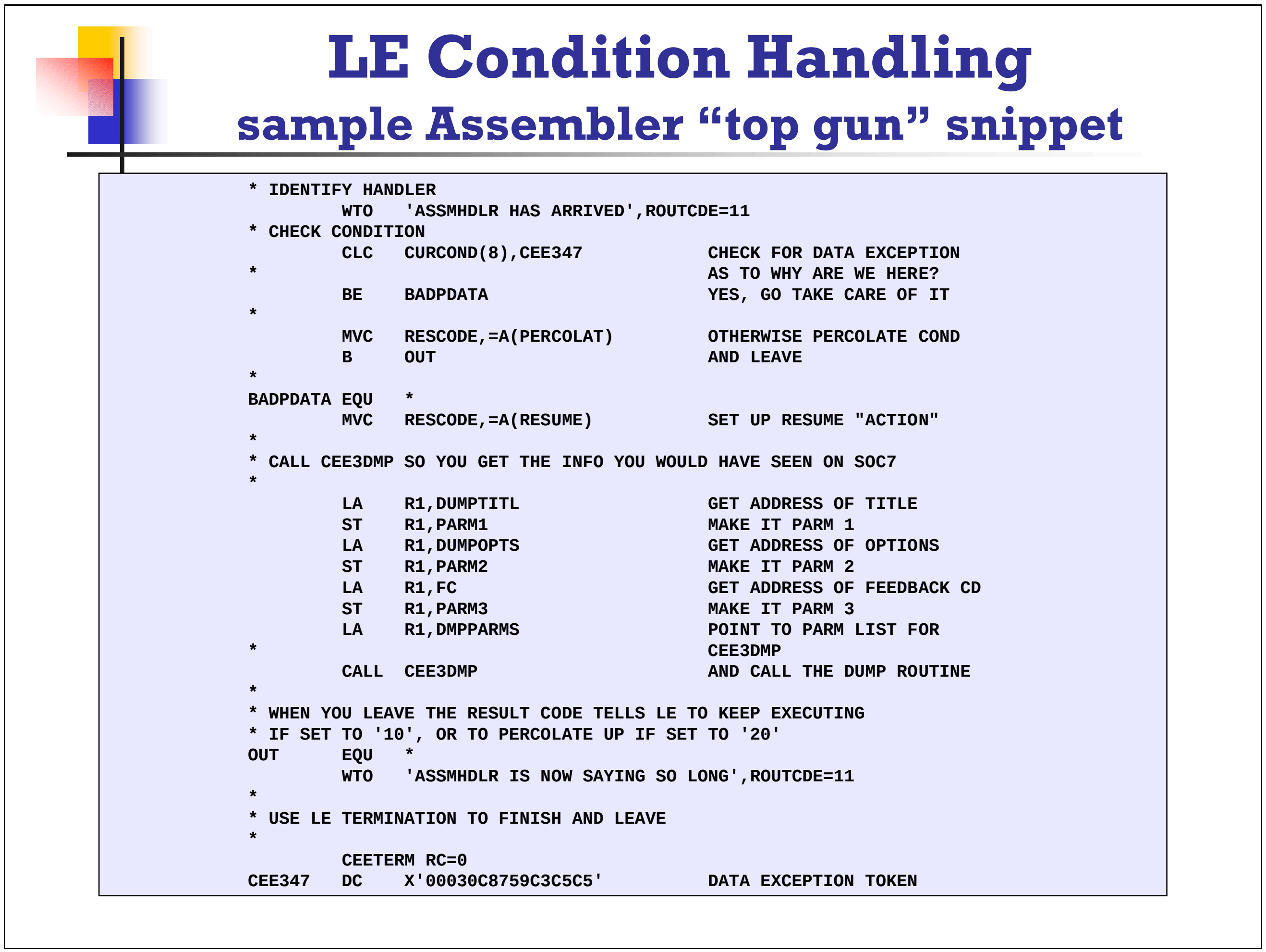
This ALWYSHDL super handler will get CONTROL FIRST, and then any other user registered handler routines will be invoked. This allows the truly super top gun of handlers to get first crack at the exception or condition before anything or anyone else touches it! You could also register another top handler, which would be invoked in the "normal" sequence of handlers, by using PARM='/USRHDLR(TOPGUN,ALWYSHDL)'.

Note that whether you have the '/' before or after user run-time options depends upon the setting of CBLOPTS(ON|OFF). CBLOPTS(ON) specifies that USER run-time options are FIRST, and the separator or initial slash identifies where the LE run-time options begin. CBLOPTS(OFF) indicates that LE run-time options PRECEDE any user parameters, and now the slash indicates the beginning of user run-times parameters.:

With CBLOPTS(ON): PARM='USEROPTS/LEOPTS'

With CBLOPTS(OFF): PARM='LEOPTS/USEROPTS'

This option ONLY APPLIES with a main program in COBOL, otherwise the normal LE order is in effect, which is PARM='LEOPTS/USEROPTS'.



LE Condition Handling

sample Assembler “top gun” snippet

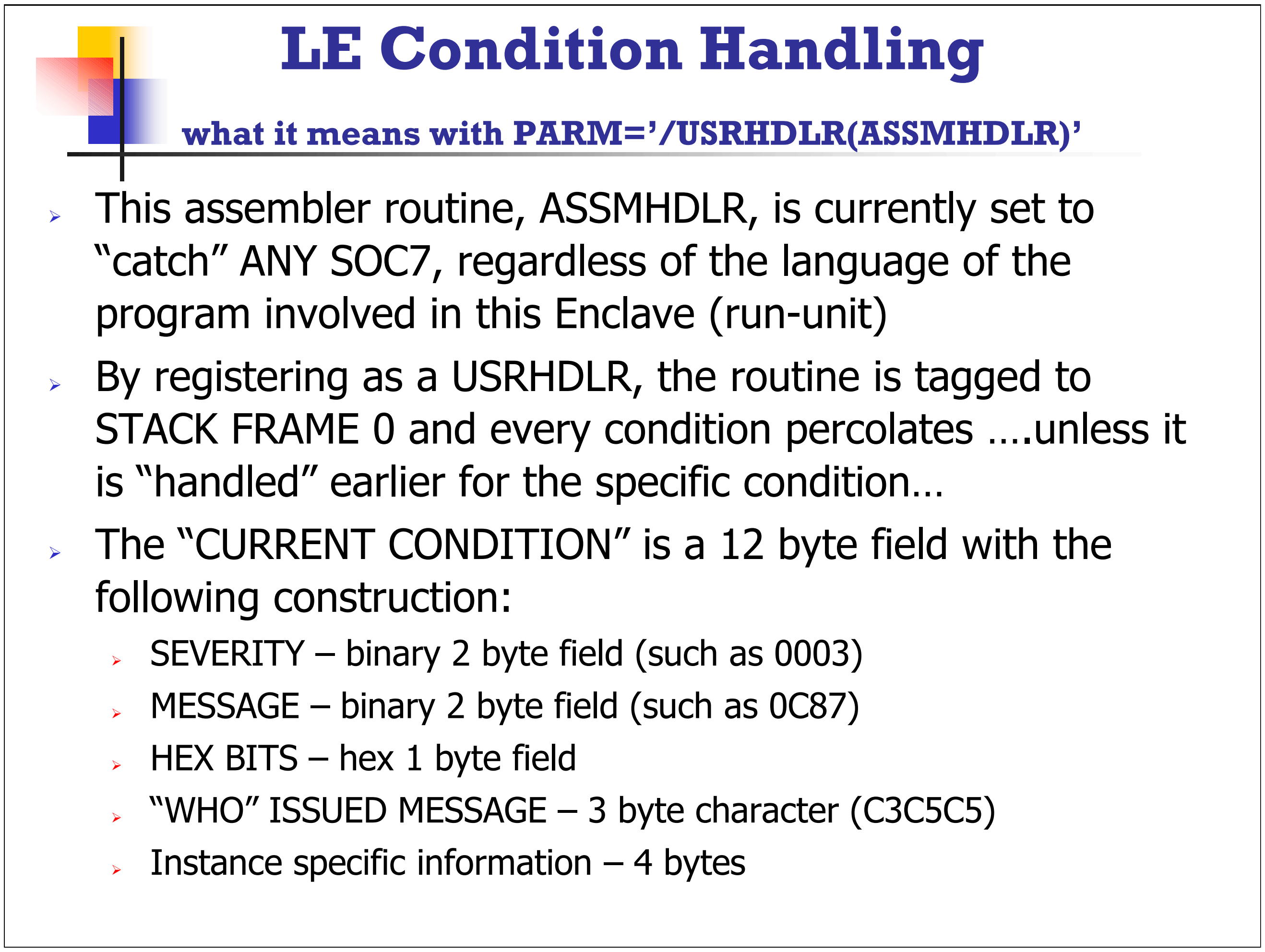
```
* IDENTIFY HANDLER
         WTO   'ASSMHDLR HAS ARRIVED',ROUTCDE=11
* CHECK CONDITION
         CLC   CURCOND(8),CEE347             CHECK FOR DATA EXCEPTION
*                                            AS TO WHY ARE WE HERE?
         BE    BADPDATA                      YES, GO TAKE CARE OF IT
*
         MVC   RESCODE,=A(PERCOLAT)          OTHERWISE PERCOLATE COND
         B     OUT                           AND LEAVE
*
BADPDATA EQU   *
         MVC   RESCODE,=A(RESUME)            SET UP RESUME "ACTION"
*
* CALL CEE3DMP SO YOU GET THE INFO YOU WOULD HAVE SEEN ON SOC7
*
         LA    R1,DUMPTITL                   GET ADDRESS OF TITLE
         ST    R1,PARM1                      MAKE IT PARM 1
         LA    R1,DUMPOPTS                   GET ADDRESS OF OPTIONS
         ST    R1,PARM2                      MAKE IT PARM 2
         LA    R1,FC                         GET ADDRESS OF FEEDBACK CD
         ST    R1,PARM3                      MAKE IT PARM 3
         LA    R1,DMPPARMS                   POINT TO PARM LIST FOR
*                                            CEE3DMP
         CALL  CEE3DMP                       AND CALL THE DUMP ROUTINE
*
* WHEN YOU LEAVE THE RESULT CODE TELLS LE TO KEEP EXECUTING
* IF SET TO '10', OR TO PERCOLATE UP IF SET TO '20'
OUT      EQU   *
         WTO   'ASSMHDLR IS NOW SAYING SO LONG',ROUTCDE=11
*
* USE LE TERMINATION TO FINISH AND LEAVE
*
         CEETERM RC=0
CEE347   DC    X'00030C8759C3C5C5'           DATA EXCEPTION TOKEN
```

The Current Condition is a 12 byte condition token that is available to be “examined” in your user handler program. The contents of this condition token indicate what just happened. The CEE347 is indicative (by name only) that a data exception has occurred. You can name these token descriptive names or use the symbolic feedback code, it’s the content that counts! The first eight bytes are the critical ones to check:

CEE347 DC X’00030C8759C3C5C5’

In order to RESUME, you send back to Language Environment the code +10, and to PERCOLATE, you would use code +20. These are only 2 of several possible return codes to Language Environment, but are the typical starter codes, either resume just past the failing instruction, or percolate to another (if it exists) handler in the stack frame sequence. If this is a handler registered via USRHDLR, then probably there are no other handler, in which case a PERCOLATE will ultimately cause the application to fail with the original condition and information just as if no handlers had ever been involved!

See the complete assembler program example in the additional handout material.

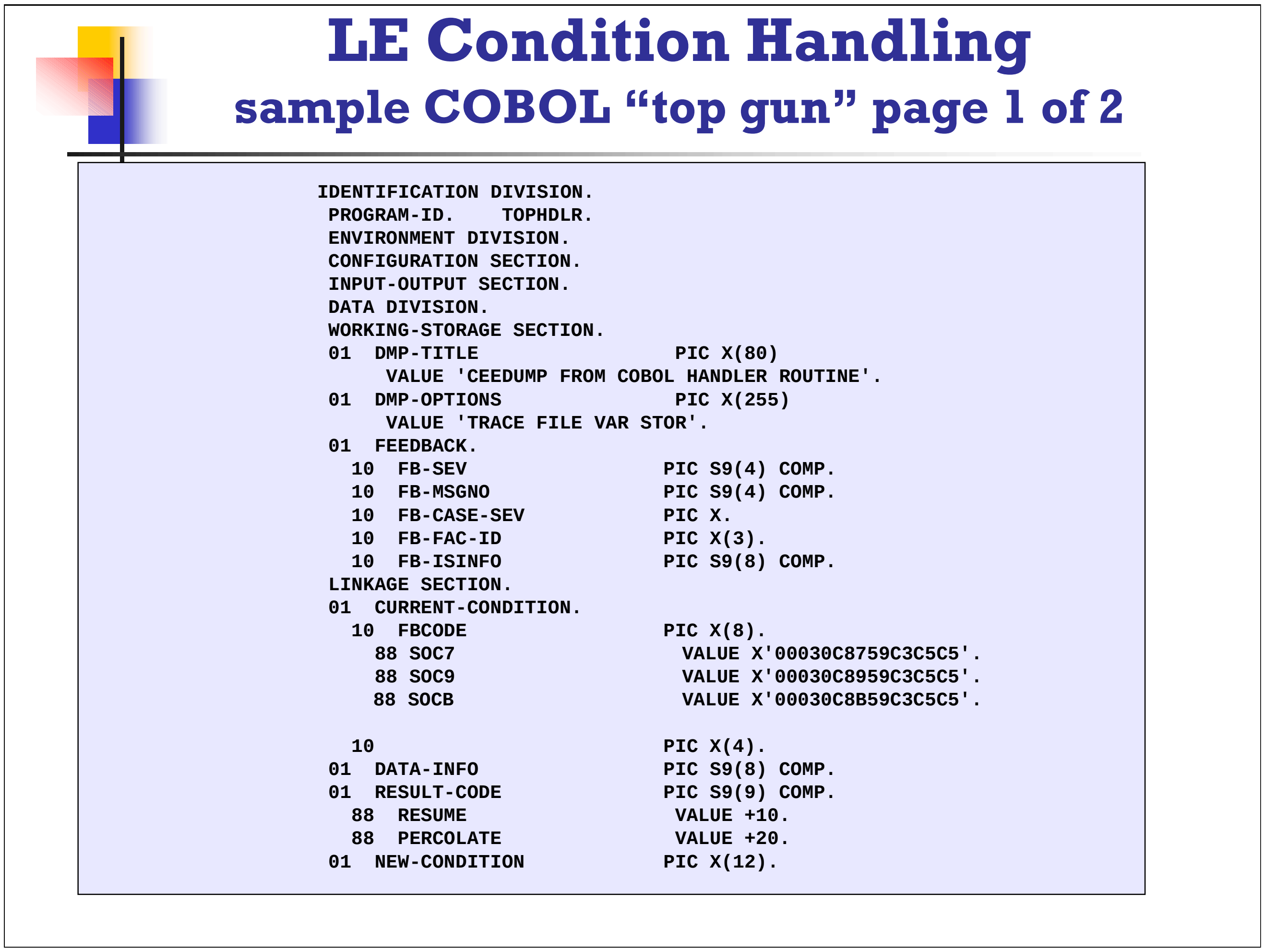


LE Condition Handling

what it means with PARM='/USRHDLR(ASSMHDLR)'

- This assembler routine, ASSMHDLR, is currently set to "catch" ANY SOC7, regardless of the language of the program involved in this Enclave (run-unit)
- By registering as a USRHDLR, the routine is tagged to STACK FRAME 0 and every condition percolates ….unless it is "handled" earlier for the specific condition…
- The "CURRENT CONDITION" is a 12 byte field with the following construction:
  - SEVERITY – binary 2 byte field (such as 0003)
  - MESSAGE – binary 2 byte field (such as 0C87)
  - HEX BITS – hex 1 byte field
  - "WHO" ISSUED MESSAGE – 3 byte character (C3C5C5)
  - Instance specific information – 4 bytes

The full program coding for ASSMHDLR is available as an additional handout. In this example the "main" program can be any language, the condition handler can also be any LE supported language, and the handler will "catch" the condition, and determine what the error is. In this example if the error is a SOC7 then the handler routine will return a code +10 to Language Environment, indicating that processing should resume at the statement following the one that caused the condition.

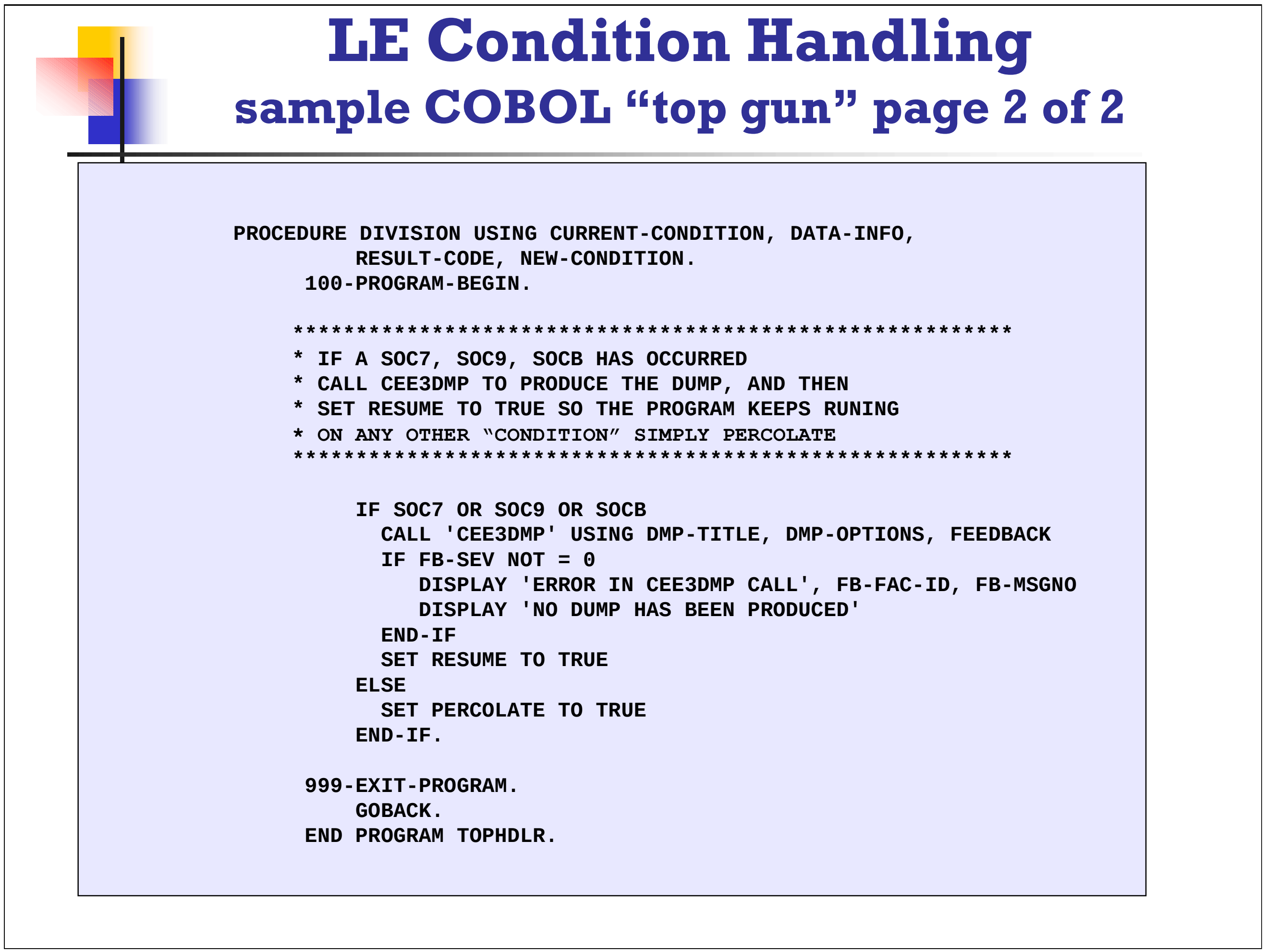


LE Condition Handling

sample COBOL “top gun” page 1 of 2

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.    TOPHDLR.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  DMP-TITLE                 PIC X(80)
      VALUE 'CEEDUMP FROM COBOL HANDLER ROUTINE'.
 01  DMP-OPTIONS               PIC X(255)
      VALUE 'TRACE FILE VAR STOR'.
 01  FEEDBACK.
   10  FB-SEV                 PIC S9(4) COMP.
   10  FB-MSGNO               PIC S9(4) COMP.
   10  FB-CASE-SEV            PIC X.
   10  FB-FAC-ID              PIC X(3).
   10  FB-ISINFO              PIC S9(8) COMP.
 LINKAGE SECTION.
 01  CURRENT-CONDITION.
   10  FBCODE                 PIC X(8).
     88 SOC7                    VALUE X'00030C8759C3C5C5'.
     88 SOC9                    VALUE X'00030C8959C3C5C5'.
     88 SOCB                    VALUE X'00030C8B59C3C5C5'.

   10                         PIC X(4).
 01  DATA-INFO                PIC S9(8) COMP.
 01  RESULT-CODE              PIC S9(9) COMP.
   88  RESUME                  VALUE +10.
   88  PERCOLATE               VALUE +20.
 01  NEW-CONDITION            PIC X(12).
```

This is a COBOL sample for a condition handler that is set to catch a SOC7, SOC9, or SOCB condition. If the error is any of these three, the handler tells LE to RESUME. Any other error is simply percolated to LE. If there are no other handlers registered (and in this case there are not), then LE will do normal program abend processing.

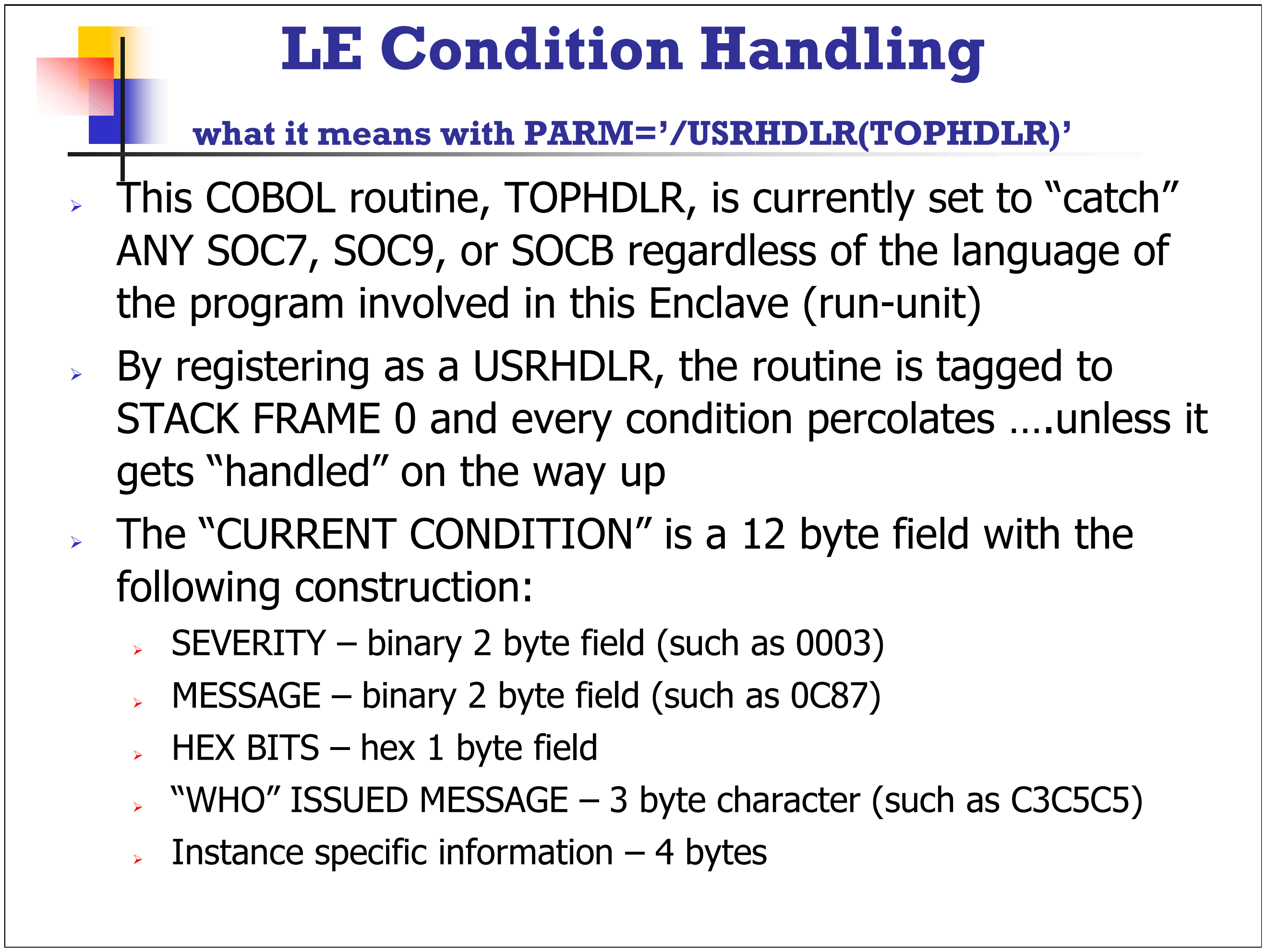


LE Condition Handling

sample COBOL “top gun” page 2 of 2

```
      PROCEDURE DIVISION USING CURRENT-CONDITION, DATA-INFO,
               RESULT-CODE, NEW-CONDITION.
           100-PROGRAM-BEGIN.

          ***********************************************************
          * IF A SOC7, SOC9, SOCB HAS OCCURRED
          * CALL CEE3DMP TO PRODUCE THE DUMP, AND THEN
          * SET RESUME TO TRUE SO THE PROGRAM KEEPS RUNING
          * ON ANY OTHER "CONDITION" SIMPLY PERCOLATE
          ***********************************************************

               IF SOC7 OR SOC9 OR SOCB
                 CALL 'CEE3DMP' USING DMP-TITLE, DMP-OPTIONS, FEEDBACK
                 IF FB-SEV NOT = 0
                    DISPLAY 'ERROR IN CEE3DMP CALL', FB-FAC-ID, FB-MSGNO
                    DISPLAY 'NO DUMP HAS BEEN PRODUCED'
                 END-IF
                 SET RESUME TO TRUE
               ELSE
                 SET PERCOLATE TO TRUE
               END-IF.

           999-EXIT-PROGRAM.
               GOBACK.
           END PROGRAM TOPHDLR.
```

Notice that the handler does a call to CEE3DMP, which will produce the CEEDUMP just as if the program had abended, but because RESUME is sent back to LE, you get the dump, but the application continues processing.

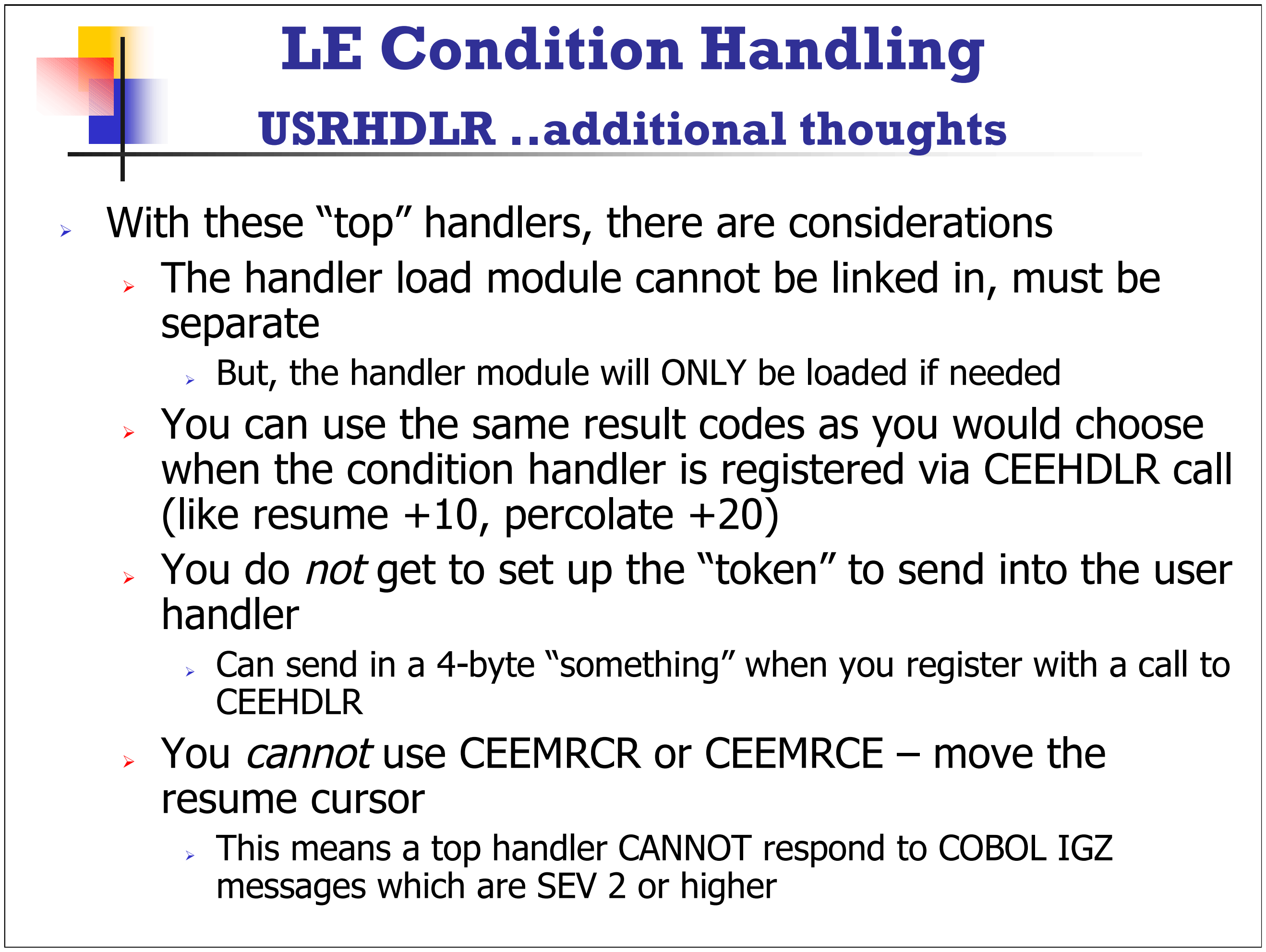


LE Condition Handling

what it means with PARM='/USRHDLR(TOPHDLR)'

- This COBOL routine, TOPHDLR, is currently set to "catch" ANY SOC7, SOC9, or SOCB regardless of the language of the program involved in this Enclave (run-unit)
- By registering as a USRHDLR, the routine is tagged to STACK FRAME 0 and every condition percolates ….unless it gets "handled" on the way up
- The "CURRENT CONDITION" is a 12 byte field with the following construction:
  - SEVERITY – binary 2 byte field (such as 0003)
  - MESSAGE – binary 2 byte field (such as 0C87)
  - HEX BITS – hex 1 byte field
  - "WHO" ISSUED MESSAGE – 3 byte character (such as C3C5C5)
  - Instance specific information – 4 bytes

The sample TOPHDLR is a complete COBOL program, albeit a simple one. This routine will create the information you would have received with CEEDUMP, but the application continues processing.

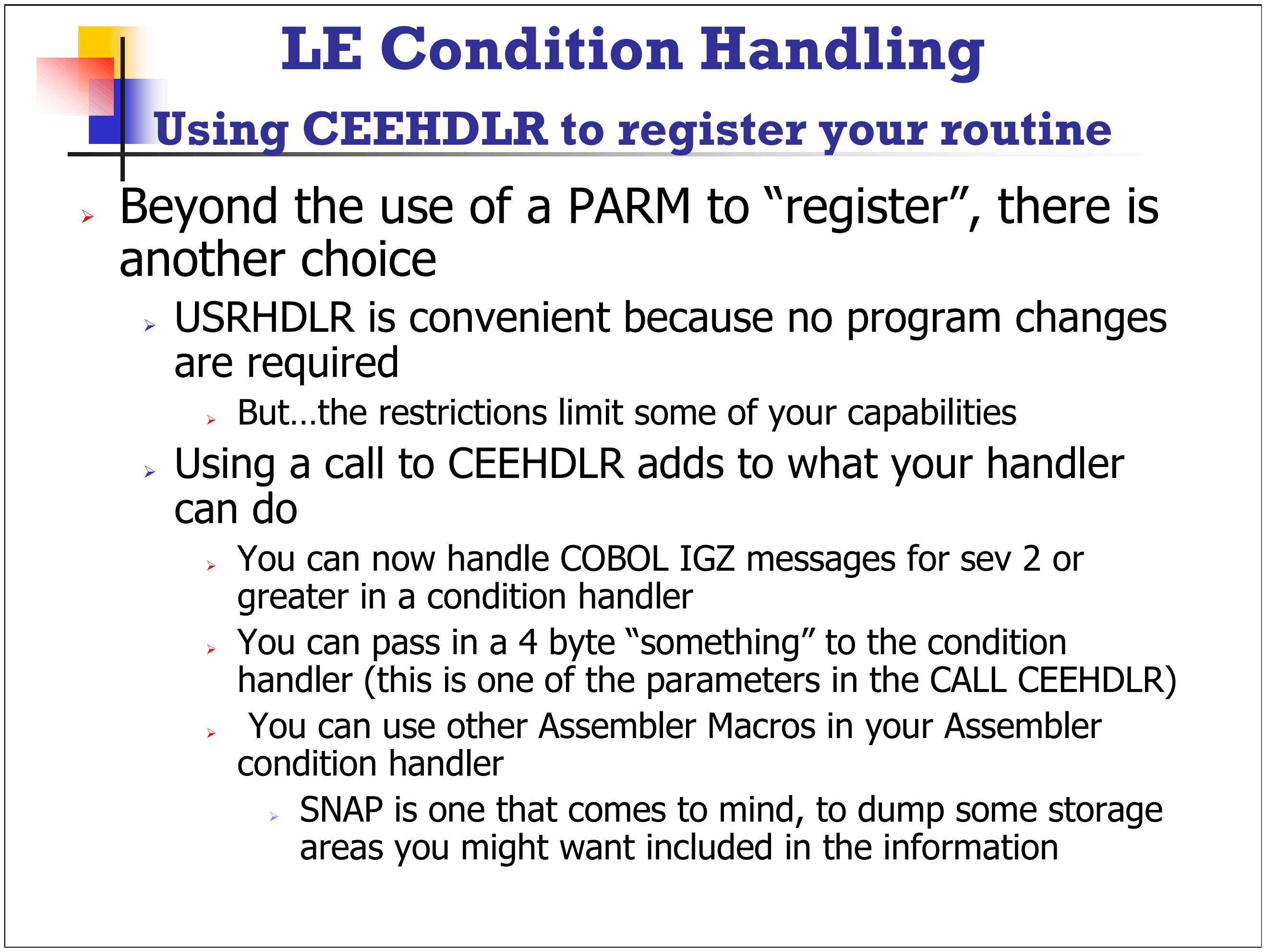


LE Condition Handling

USRHDLR ..additional thoughts

- With these "top" handlers, there are considerations
  - The handler load module cannot be linked in, must be separate
    - But, the handler module will ONLY be loaded if needed
  - You can use the same result codes as you would choose when the condition handler is registered via CEEHDLR call (like resume +10, percolate +20)
  - You do *not* get to set up the "token" to send into the user handler
    - Can send in a 4-byte "something" when you register with a call to CEEHDLR
  - You *cannot* use CEEMRCR or CEEMRCE – move the resume cursor
    - This means a top handler CANNOT respond to COBOL IGZ messages which are SEV 2 or higher

It would typically be good practice to have all the condition handlers as "dynamic" modules. You would then only get them loaded and executed in the event you have a program "condition". With the handlers registered via the USRHDLR run-time option, you must have the modules as separate. With handlers registered via CEEHDLR you can either have the condition handler program dynamically linked, or statically linked.



LE Condition Handling

Using CEEHDLR to register your routine

- Beyond the use of a PARM to "register", there is another choice
  - USRHDLR is convenient because no program changes are required
    - But…the restrictions limit some of your capabilities
  - Using a call to CEEHDLR adds to what your handler can do
    - You can now handle COBOL IGZ messages for sev 2 or greater in a condition handler
    - You can pass in a 4 byte "something" to the condition handler (this is one of the parameters in the CALL CEEHDLR)
    - You can use other Assembler Macros in your Assembler condition handler
      - SNAP is one that comes to mind, to dump some storage areas you might want included in the information

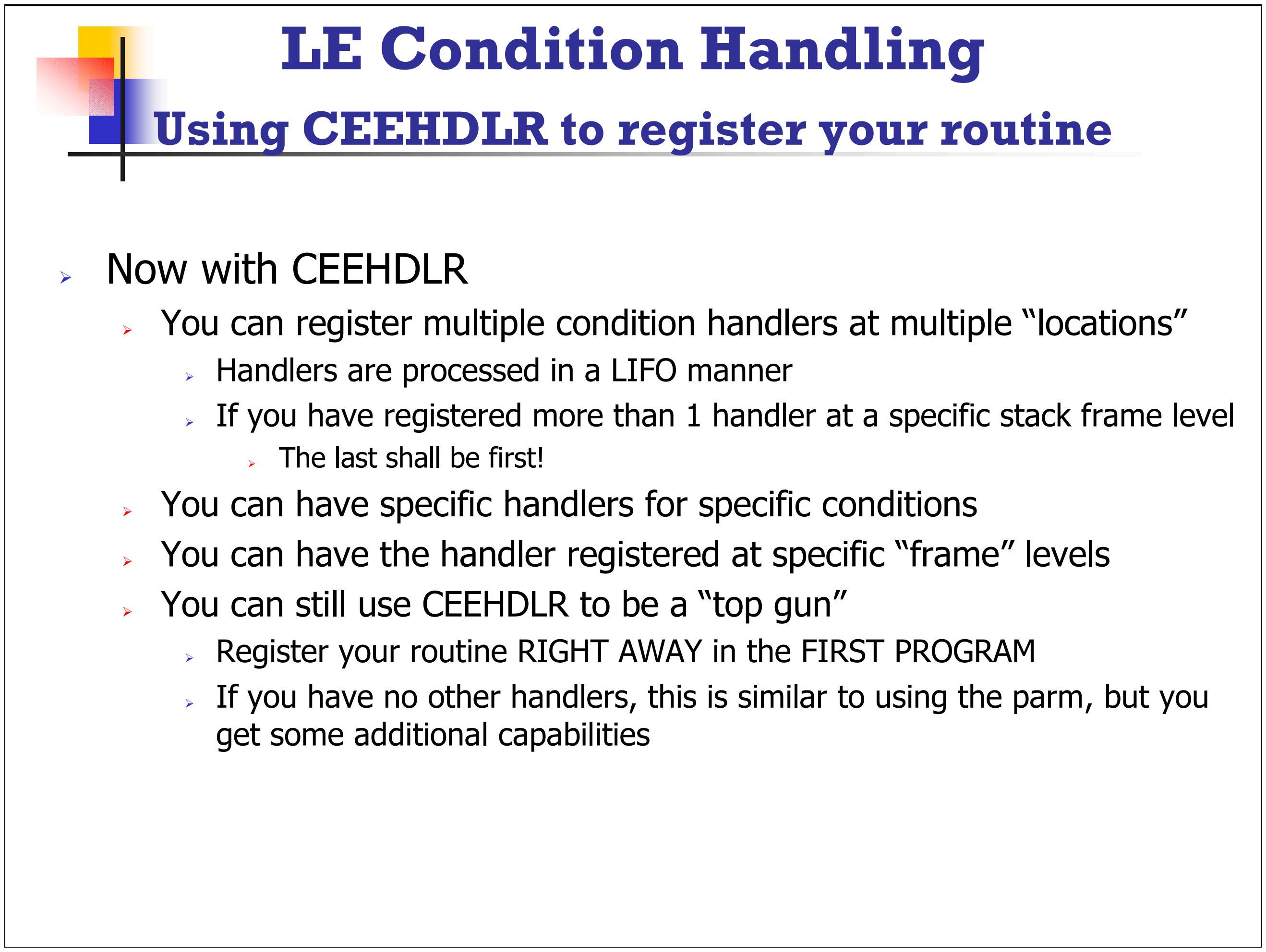
With the call to CEEHDLR there are 4 parameters.

•1st parameter is a pointer with the name of the Condition Handler Program,

•2nd parameter is a 4-byte something, perhaps a pointer to a structure, which can be sent INTO the condition handler,

•3rd parameter is the result code to send back to LE,

•4th parameter is a new-condition, should the handler have code to CHANGE the condition from the current-condition.

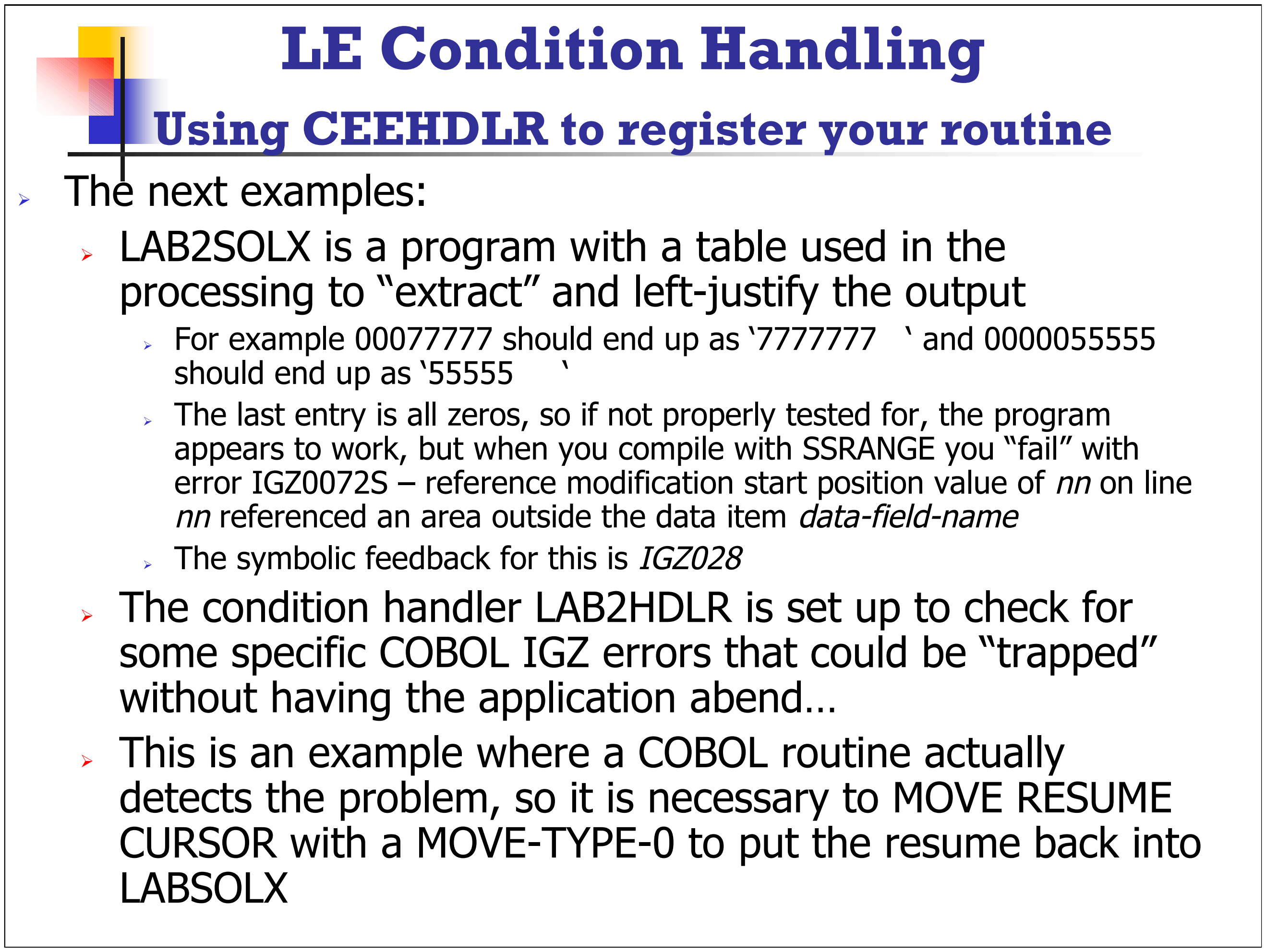


LE Condition Handling

Using CEEHDLR to register your routine

- Now with CEEHDLR
  - You can register multiple condition handlers at multiple “locations”
    - Handlers are processed in a LIFO manner
    - If you have registered more than 1 handler at a specific stack frame level
      - The last shall be first!
  - You can have specific handlers for specific conditions
  - You can have the handler registered at specific “frame” levels
  - You can still use CEEHDLR to be a “top gun”
    - Register your routine RIGHT AWAY in the FIRST PROGRAM
    - If you have no other handlers, this is similar to using the parm, but you get some additional capabilities

CEEHDLR can be called to register AS MANY CONDITION HANDLERS as you might need. This allows you to have VERY SPECIFIC condition handlers, and you can register each one independently, and LE will find them all! The condition handlers will be invoked in the reverse order they are registered.

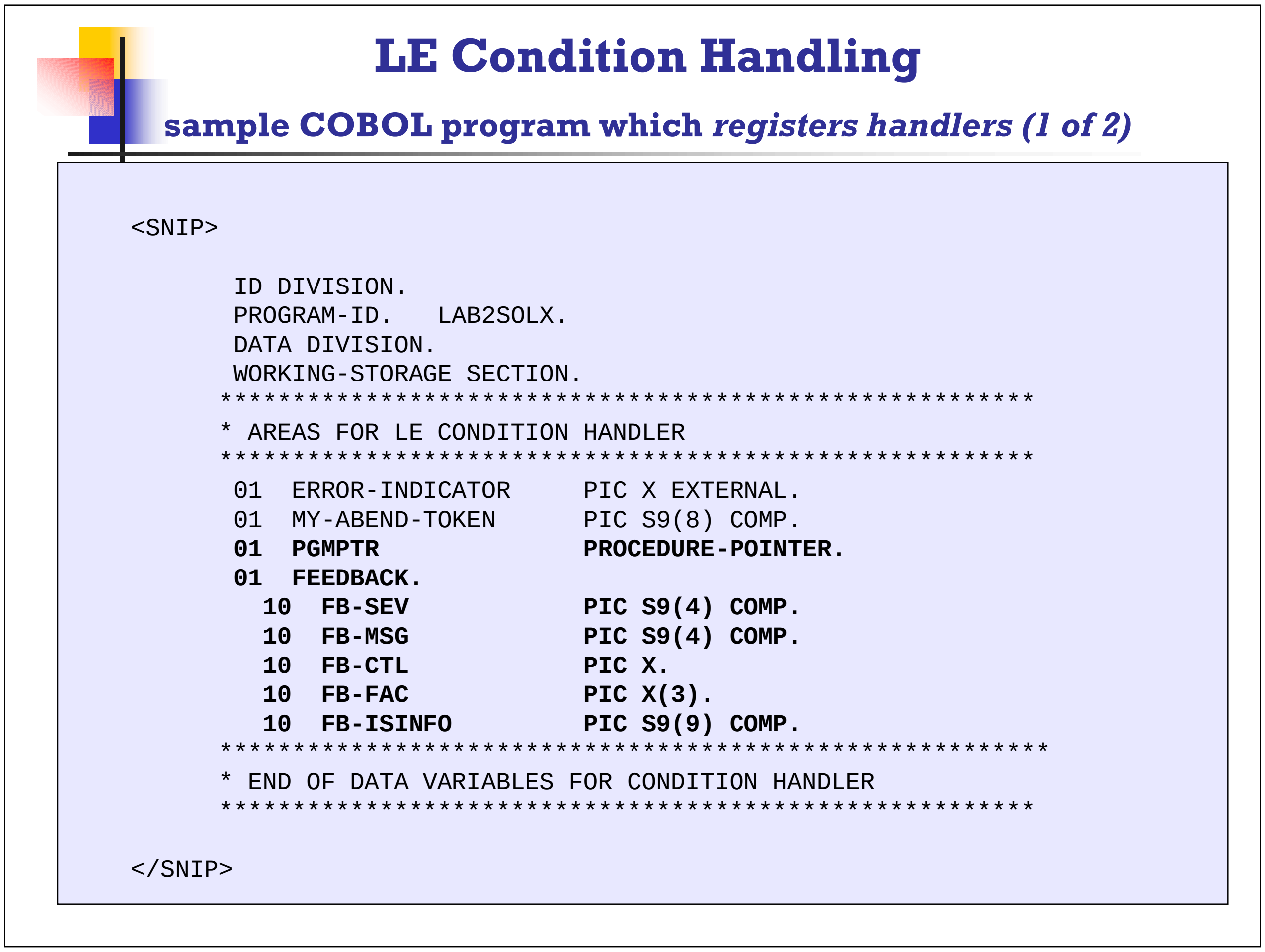


LE Condition Handling

Using CEEHDLR to register your routine

- The next examples:
  - LAB2SOLX is a program with a table used in the processing to “extract” and left-justify the output
    - For example 00077777 should end up as ‘7777777   ‘ and 0000055555 should end up as ‘55555   ‘
    - The last entry is all zeros, so if not properly tested for, the program appears to work, but when you compile with SSRANGE you “fail” with error IGZ0072S – reference modification start position value of *nn* on line *nn* referenced an area outside the data item *data-field-name*
    - The symbolic feedback for this is *IGZ028*
  - The condition handler LAB2HDLR is set up to check for some specific COBOL IGZ errors that could be “trapped” without having the application abend…
  - This is an example where a COBOL routine actually detects the problem, so it is necessary to MOVE RESUME CURSOR with a MOVE-TYPE-0 to put the resume back into LABSOLX

See the programming sample for the LAB2SOLX and LAB2HDLR in the additional handout material. The following Pages will only show snippets from the code.

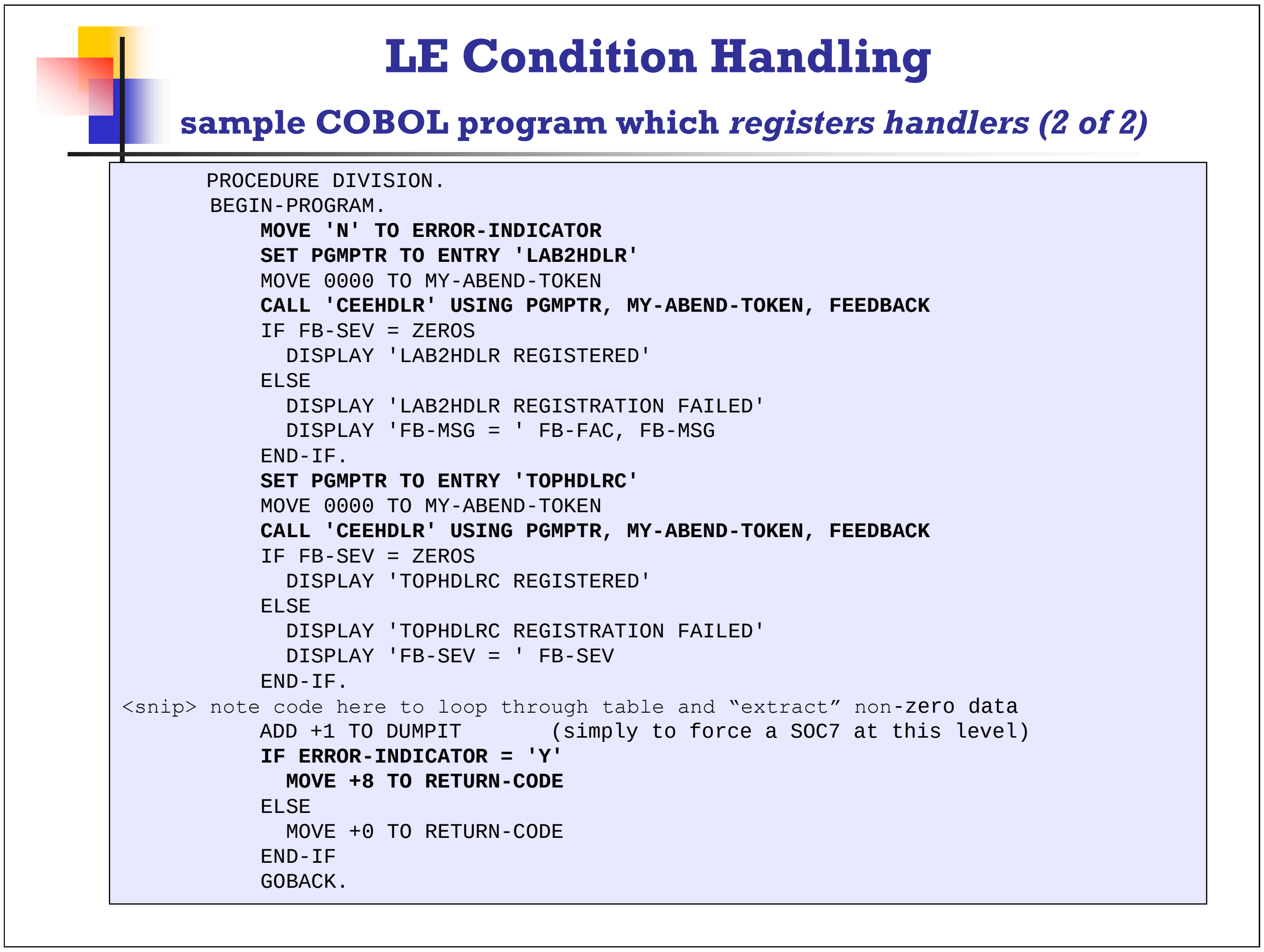


LE Condition Handling

sample COBOL program which *registers handlers (1 of 2)*

```
<SNIP>

       ID DIVISION.
       PROGRAM-ID.   LAB2SOLX.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      ********************************************************
      * AREAS FOR LE CONDITION HANDLER
      ********************************************************
       01  ERROR-INDICATOR     PIC X EXTERNAL.
       01  MY-ABEND-TOKEN      PIC S9(8) COMP.
       01  PGMPTR              PROCEDURE-POINTER.
       01  FEEDBACK.
         10  FB-SEV            PIC S9(4) COMP.
         10  FB-MSG            PIC S9(4) COMP.
         10  FB-CTL            PIC X.
         10  FB-FAC            PIC X(3).
         10  FB-ISINFO         PIC S9(9) COMP.
      *********************************************************
      * END OF DATA VARIABLES FOR CONDITION HANDLER
      ********************************************************

</SNIP>
```

The complete program sample is provided as an additional supplemental handout. The PGMPTR PROCEDURE-POINTER data item will be set up with the name of the condition handler for the call to CEEHDLR. MY-ABEND-TOKEN is an 4 byte item that can be SENT INTO the condition handler (it is a one-way ticket only). Also notice that the FEEDBACK structure is a 12-byte area with information that contains the RESULTS from EVERY call to a language environment routine. The layout of this feedback area is consistent regardless of the program/language/facility, and can be used to check whether the call to the Language Environment routine was successful. If the 1st halfword is NOT zeros, there was some problem. The problem will be indicated by the 2nd halfword, which will be the binary representation of the error message. The FB-FAC will indicate “who issued” the message. Since in this example we are calling an LE routine, the FAC will contain CEE.

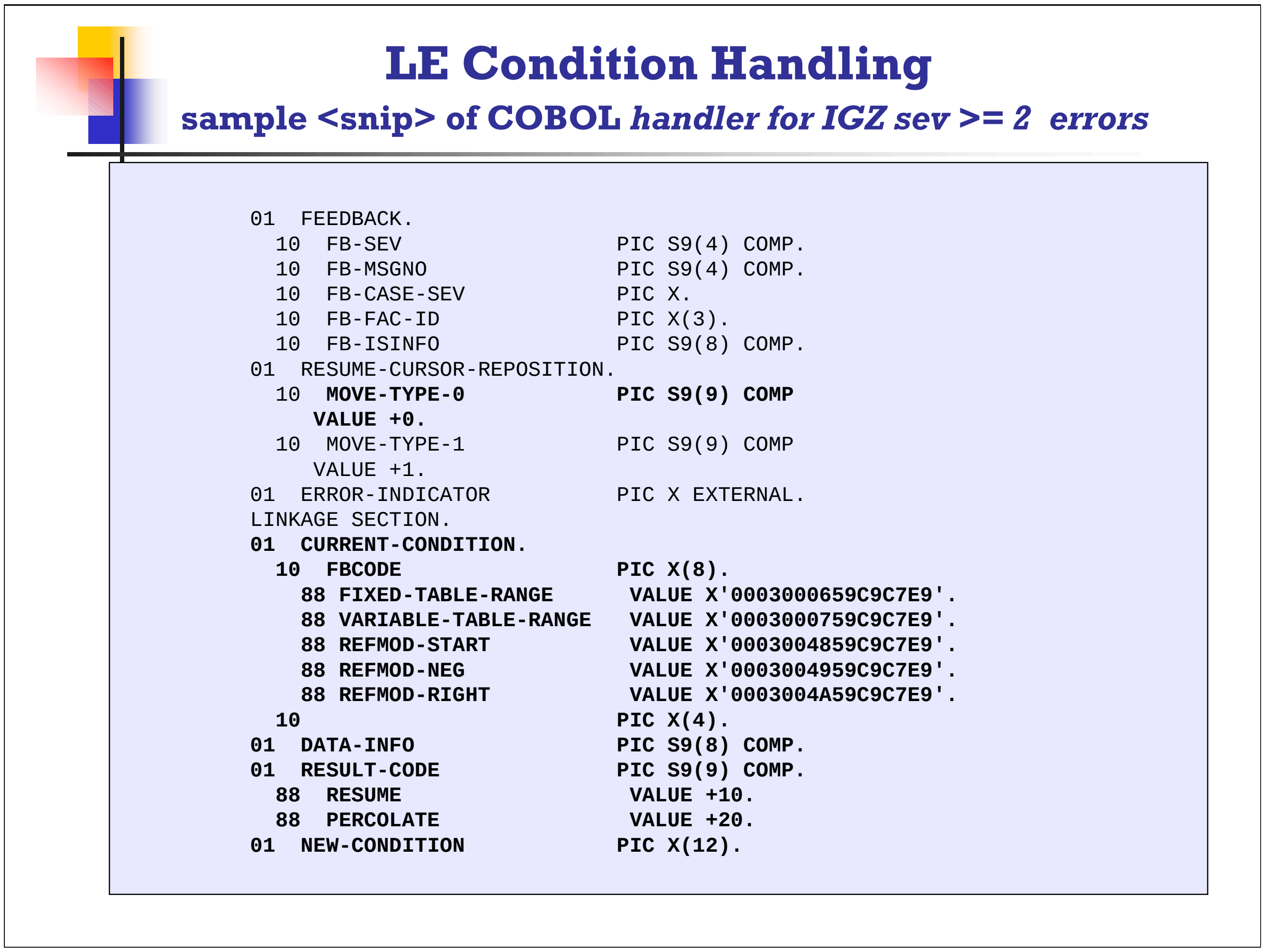


LE Condition Handling

sample COBOL program which *registers handlers (2 of 2)*

```
       PROCEDURE DIVISION.
       BEGIN-PROGRAM.
           MOVE 'N' TO ERROR-INDICATOR
           SET PGMPTR TO ENTRY 'LAB2HDLR'
           MOVE 0000 TO MY-ABEND-TOKEN
           CALL 'CEEHDLR' USING PGMPTR, MY-ABEND-TOKEN, FEEDBACK
           IF FB-SEV = ZEROS
             DISPLAY 'LAB2HDLR REGISTERED'
           ELSE
             DISPLAY 'LAB2HDLR REGISTRATION FAILED'
             DISPLAY 'FB-MSG = ' FB-FAC, FB-MSG
           END-IF.
           SET PGMPTR TO ENTRY 'TOPHDLRC'
           MOVE 0000 TO MY-ABEND-TOKEN
           CALL 'CEEHDLR' USING PGMPTR, MY-ABEND-TOKEN, FEEDBACK
           IF FB-SEV = ZEROS
             DISPLAY 'TOPHDLRC REGISTERED'
           ELSE
             DISPLAY 'TOPHDLRC REGISTRATION FAILED'
             DISPLAY 'FB-SEV = ' FB-SEV
           END-IF.
<snip> note code here to loop through table and “extract” non-zero data
           ADD +1 TO DUMPIT       (simply to force a SOC7 at this level)
           IF ERROR-INDICATOR = 'Y'
             MOVE +8 TO RETURN-CODE
           ELSE
             MOVE +0 TO RETURN-CODE
           END-IF
           GOBACK.
```

This program is set up to demonstrate two different capabilities of Condition Handling. The first “error” that occurs is a COBOL detected error. The program loops through the table data to eliminate leading zeroes and display the “rest of the field”. The last entry is all zeros, so counting the leading zeroes gives a count of 10, then when you attempt to move the rest of the field to DDANO-OUT, you are actually referencing a position OUTSIDE the range of the field. This “error” is only detected when you compile with SSRANGE, and execute with CHECK(ON). Since a COBOL routine detects the error, the condition handler cannot simply resume execution, because the resume cursor is pointing to the COBOL routine. This is an example where you should use the move resume cursor capability, to move the resume cursor “back” one level, which will put the resume cursor back in your COBOL application program and you can correctly and properly resume execution! Note also that the ERROR-INDICATOR is defined as an EXTERNAL data item in both the COBOL application program and in the COBOL condition handler. This allows the condition handler to “set” an indicator that can be checked in the main COBOL program so that the RETURN-CODE can indicate some “problem” occurred. Using a little bit of added logic in the COBOL main program, and coordinating with the condition handler, allows some additional information to be provided. Contrast this with the super condition handler, using USRHDLR, which can trap the SOC7, but NOT this type of COBOL error. Additionally using 2 COBOL programs and cross-communicating allow the program to indicate via JCL RETURN-CODE that something was amiss!

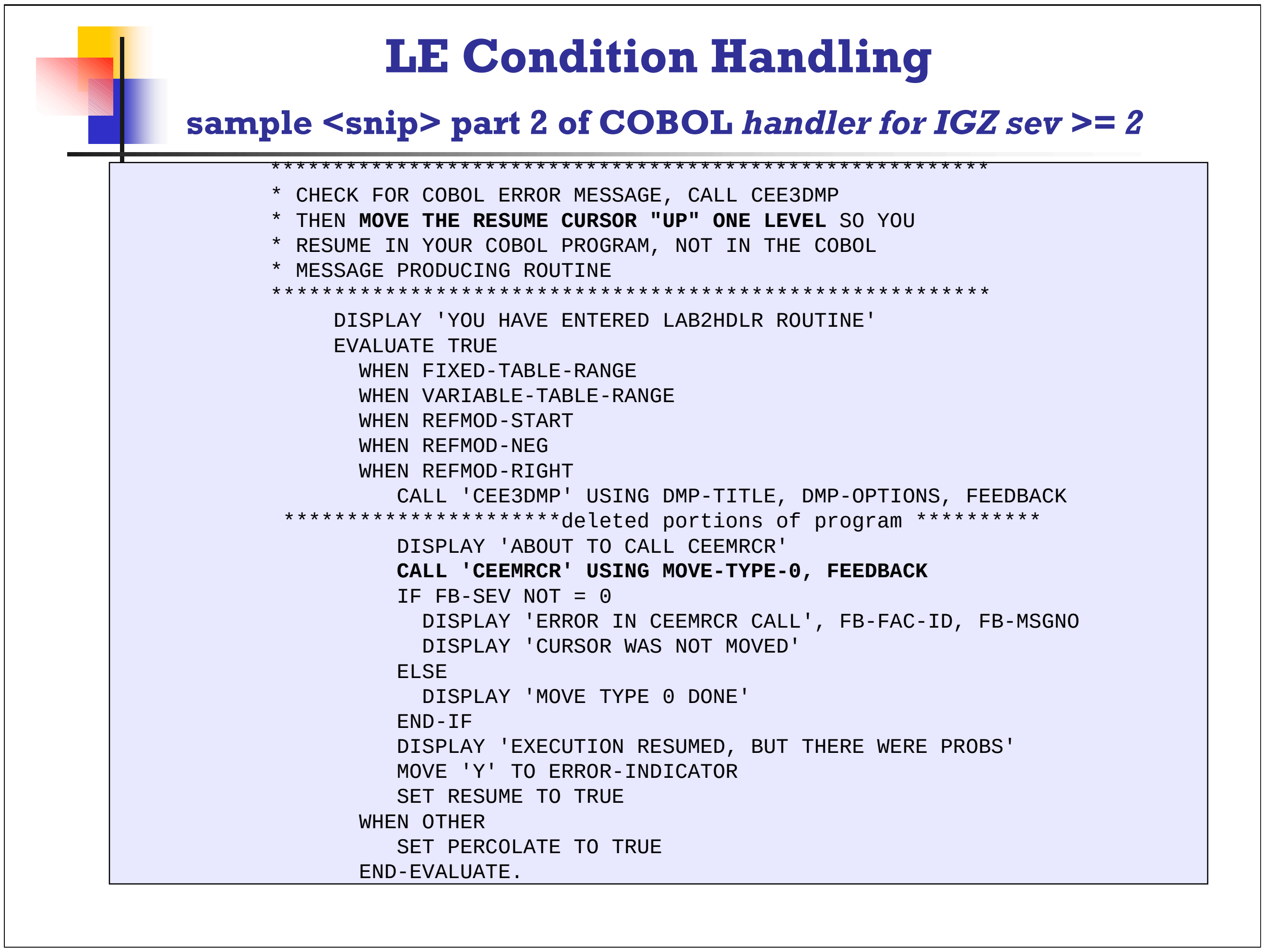


LE Condition Handling

sample <snip> of COBOL *handler for IGZ sev >= 2 errors*

```
01  FEEDBACK.
  10  FB-SEV                 PIC S9(4) COMP.
  10  FB-MSGNO               PIC S9(4) COMP.
  10  FB-CASE-SEV            PIC X.
  10  FB-FAC-ID              PIC X(3).
  10  FB-ISINFO              PIC S9(8) COMP.
01  RESUME-CURSOR-REPOSITION.
  10  MOVE-TYPE-0            PIC S9(9) COMP
     VALUE +0.
  10  MOVE-TYPE-1            PIC S9(9) COMP
     VALUE +1.
01  ERROR-INDICATOR          PIC X EXTERNAL.
LINKAGE SECTION.
01  CURRENT-CONDITION.
  10  FBCODE                 PIC X(8).
    88 FIXED-TABLE-RANGE      VALUE X'0003000659C9C7E9'.
    88 VARIABLE-TABLE-RANGE   VALUE X'0003000759C9C7E9'.
    88 REFMOD-START           VALUE X'0003004859C9C7E9'.
    88 REFMOD-NEG             VALUE X'0003004959C9C7E9'.
    88 REFMOD-RIGHT           VALUE X'0003004A59C9C7E9'.
  10                         PIC X(4).
01  DATA-INFO                PIC S9(8) COMP.
01  RESULT-CODE              PIC S9(9) COMP.
  88  RESUME                  VALUE +10.
  88  PERCOLATE               VALUE +20.
01  NEW-CONDITION            PIC X(12).
```

This is a portion of the COBOL coding in LAB2HDLR, which is available in its entirety as a separate handout. Notice that the feedback code is now checking for some specific COBOL IGZ type errors, related to table range or reference modification positions being outside the "range" or size of the field. The symbolic feedback codes are actually: IGZ006 – from message IGZ0006S, the reference to table *table-name* by verb number *verb-number on line line-number* addressed an area outside the region of the table, IGZ007 – from message IGZ0007S, the reference to variable-length group *group-name* by verb number *verb-number* on line *line-number* addressed an area outside the maximum length of the group. IGZ0072S, A reference modification start position value of *reference-modification-value* on line *line-number* referenced an area outside the region of data item *data-item,* IGZ0073S, symbolic feedback IGZ028, etc.

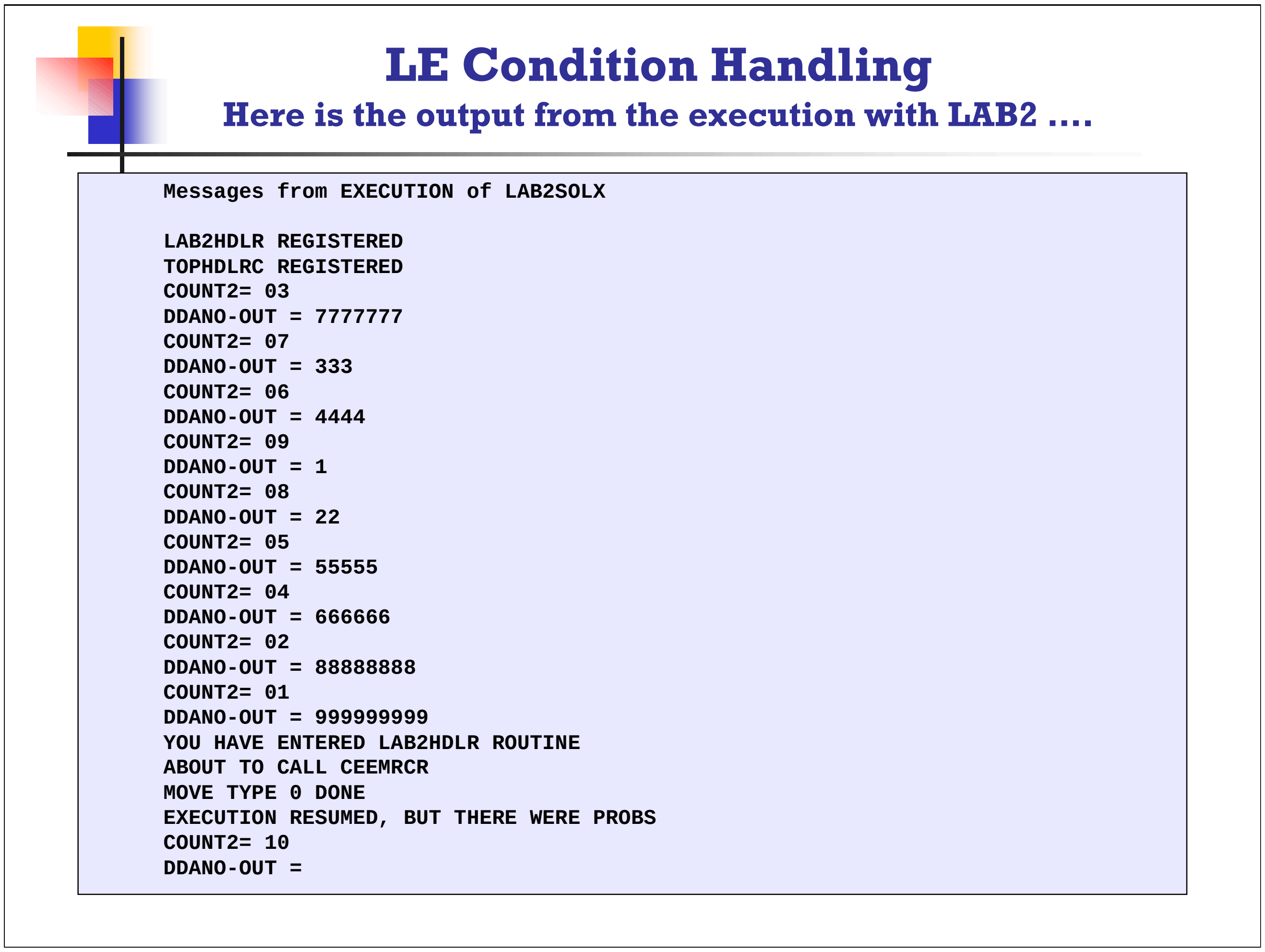


LE Condition Handling

sample <snip> part 2 of COBOL *handler for IGZ sev >= 2*

```
*********************************************************
* CHECK FOR COBOL ERROR MESSAGE, CALL CEE3DMP
* THEN MOVE THE RESUME CURSOR "UP" ONE LEVEL SO YOU
* RESUME IN YOUR COBOL PROGRAM, NOT IN THE COBOL
* MESSAGE PRODUCING ROUTINE
*********************************************************
     DISPLAY 'YOU HAVE ENTERED LAB2HDLR ROUTINE'
     EVALUATE TRUE
       WHEN FIXED-TABLE-RANGE
       WHEN VARIABLE-TABLE-RANGE
       WHEN REFMOD-START
       WHEN REFMOD-NEG
       WHEN REFMOD-RIGHT
          CALL 'CEE3DMP' USING DMP-TITLE, DMP-OPTIONS, FEEDBACK
 *********************deleted portions of program **********
          DISPLAY 'ABOUT TO CALL CEEMRCR'
          CALL 'CEEMRCR' USING MOVE-TYPE-0, FEEDBACK
          IF FB-SEV NOT = 0
            DISPLAY 'ERROR IN CEEMRCR CALL', FB-FAC-ID, FB-MSGNO
            DISPLAY 'CURSOR WAS NOT MOVED'
          ELSE
            DISPLAY 'MOVE TYPE 0 DONE'
          END-IF
          DISPLAY 'EXECUTION RESUMED, BUT THERE WERE PROBS'
          MOVE 'Y' TO ERROR-INDICATOR
          SET RESUME TO TRUE
       WHEN OTHER
          SET PERCOLATE TO TRUE
       END-EVALUATE.
```

Since the checking is for a COBOL error message, a simple RESUME is not valid. The resume cursor would be pointing to the next instruction IN THE COBOL ERROR ROUTINE, not in your application program! The MOVE RESUME CURSOR moves the RESUME location to the point in the "up one level" program, in this case, the COBOL PROGRAM, not the COBOL ERROR ROUTINE! The BOTTOM LINE MESSAGE here is, if the message is an IGZ message, you must use the MOVE RESUME CURSOR +0 to move the cursor UP 1 level in order to RESUME in YOUR program. You can get a DIVIDE BY ZERO that is a CEE message, OR, you might get one from COBOL if COBOL is the detector of the problem. If the divide by zero is detected by COBOL then you need to use a technique like the one shown here – and this also means you must REGISTER the CONDITION HANDLER IN YOUR PROGRAM, you cannot simply RESUME without moving the resume cursor!

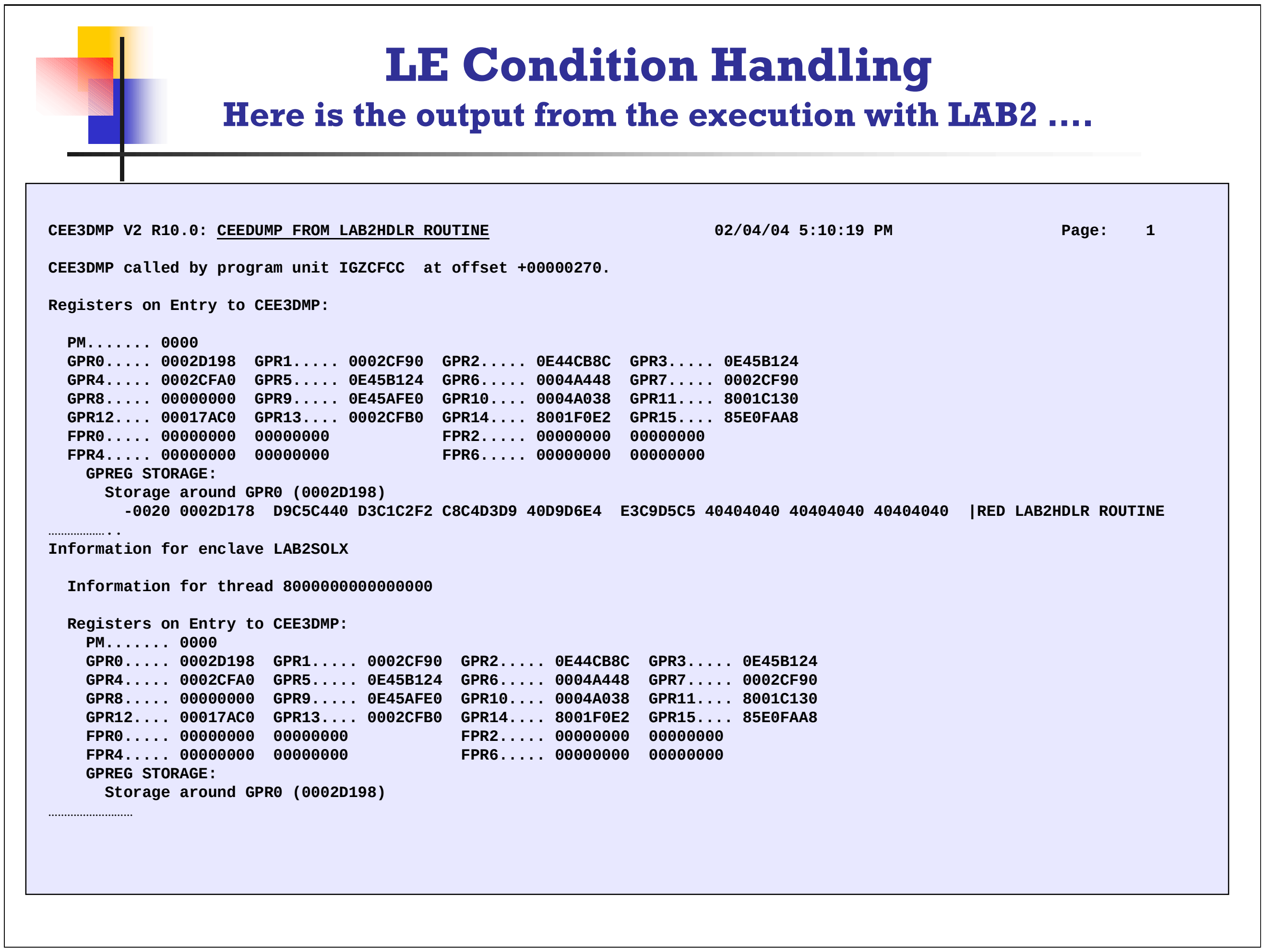


LE Condition Handling

Here is the output from the execution with LAB2 ….

```
Messages from EXECUTION of LAB2SOLX

LAB2HDLR REGISTERED
TOPHDLRC REGISTERED
COUNT2= 03
DDANO-OUT = 7777777
COUNT2= 07
DDANO-OUT = 333
COUNT2= 06
DDANO-OUT = 4444
COUNT2= 09
DDANO-OUT = 1
COUNT2= 08
DDANO-OUT = 22
COUNT2= 05
DDANO-OUT = 55555
COUNT2= 04
DDANO-OUT = 666666
COUNT2= 02
DDANO-OUT = 88888888
COUNT2= 01
DDANO-OUT = 999999999
YOU HAVE ENTERED LAB2HDLR ROUTINE
ABOUT TO CALL CEEMRCR
MOVE TYPE 0 DONE
EXECUTION RESUMED, BUT THERE WERE PROBS
COUNT2= 10
DDANO-OUT =
```

This output shows the sequence that the 2 handler routines were “registered”, so there are 2 handle cursors at this stack frame, the application displays the information as it loops through the table, then the last entry in the table has all zeros to create the “condition” of a COBOL detected reference modification “out of range” error, so the MOVE TYPE 0 moves the resume cursor up 1 frame, and execution continues.

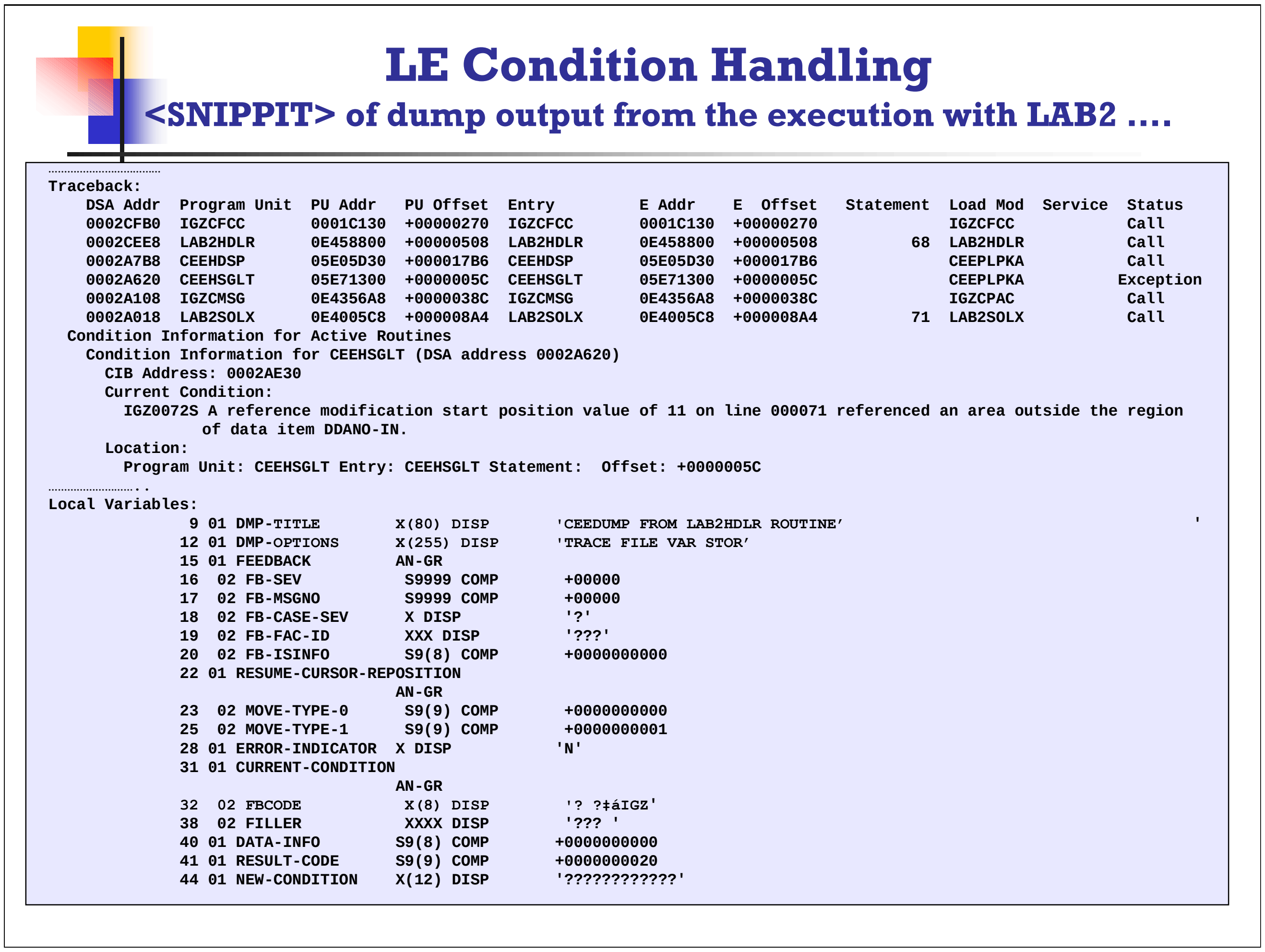


LE Condition Handling

Here is the output from the execution with LAB2 ….

```
CEE3DMP V2 R10.0: CEEDUMP FROM LAB2HDLR ROUTINE                     02/04/04 5:10:19 PM                  Page:    1

CEE3DMP called by program unit IGZCFCC  at offset +00000270.

Registers on Entry to CEE3DMP:

  PM....... 0000
  GPR0..... 0002D198  GPR1..... 0002CF90  GPR2..... 0E44CB8C  GPR3..... 0E45B124
  GPR4..... 0002CFA0  GPR5..... 0E45B124  GPR6..... 0004A448  GPR7..... 0002CF90
  GPR8..... 00000000  GPR9..... 0E45AFE0  GPR10.... 0004A038  GPR11.... 8001C130
  GPR12.... 00017AC0  GPR13.... 0002CFB0  GPR14.... 8001F0E2  GPR15.... 85E0FAA8
  FPR0..... 00000000  00000000            FPR2..... 00000000  00000000
  FPR4..... 00000000  00000000            FPR6..... 00000000  00000000
    GPREG STORAGE:
      Storage around GPR0 (0002D198)
        -0020 0002D178  D9C5C440 D3C1C2F2 C8C4D3D9 40D9D6E4  E3C9D5C5 40404040 40404040 40404040  |RED LAB2HDLR ROUTINE
……………….. .
Information for enclave LAB2SOLX

  Information for thread 8000000000000000

  Registers on Entry to CEE3DMP:
    PM....... 0000
    GPR0..... 0002D198  GPR1..... 0002CF90  GPR2..... 0E44CB8C  GPR3..... 0E45B124
    GPR4..... 0002CFA0  GPR5..... 0E45B124  GPR6..... 0004A448  GPR7..... 0002CF90
    GPR8..... 00000000  GPR9..... 0E45AFE0  GPR10.... 0004A038  GPR11.... 8001C130
    GPR12.... 00017AC0  GPR13.... 0002CFB0  GPR14.... 8001F0E2  GPR15.... 85E0FAA8
    FPR0..... 00000000  00000000            FPR2..... 00000000  00000000
    FPR4..... 00000000  00000000            FPR6..... 00000000  00000000
    GPREG STORAGE:
      Storage around GPR0 (0002D198)
………………………
```

This is a portion of the dump that resulted because LAB2HDLR was “invoked”, and as part of the processing this handler routine did a CALL CEE3DMP to produce this information.

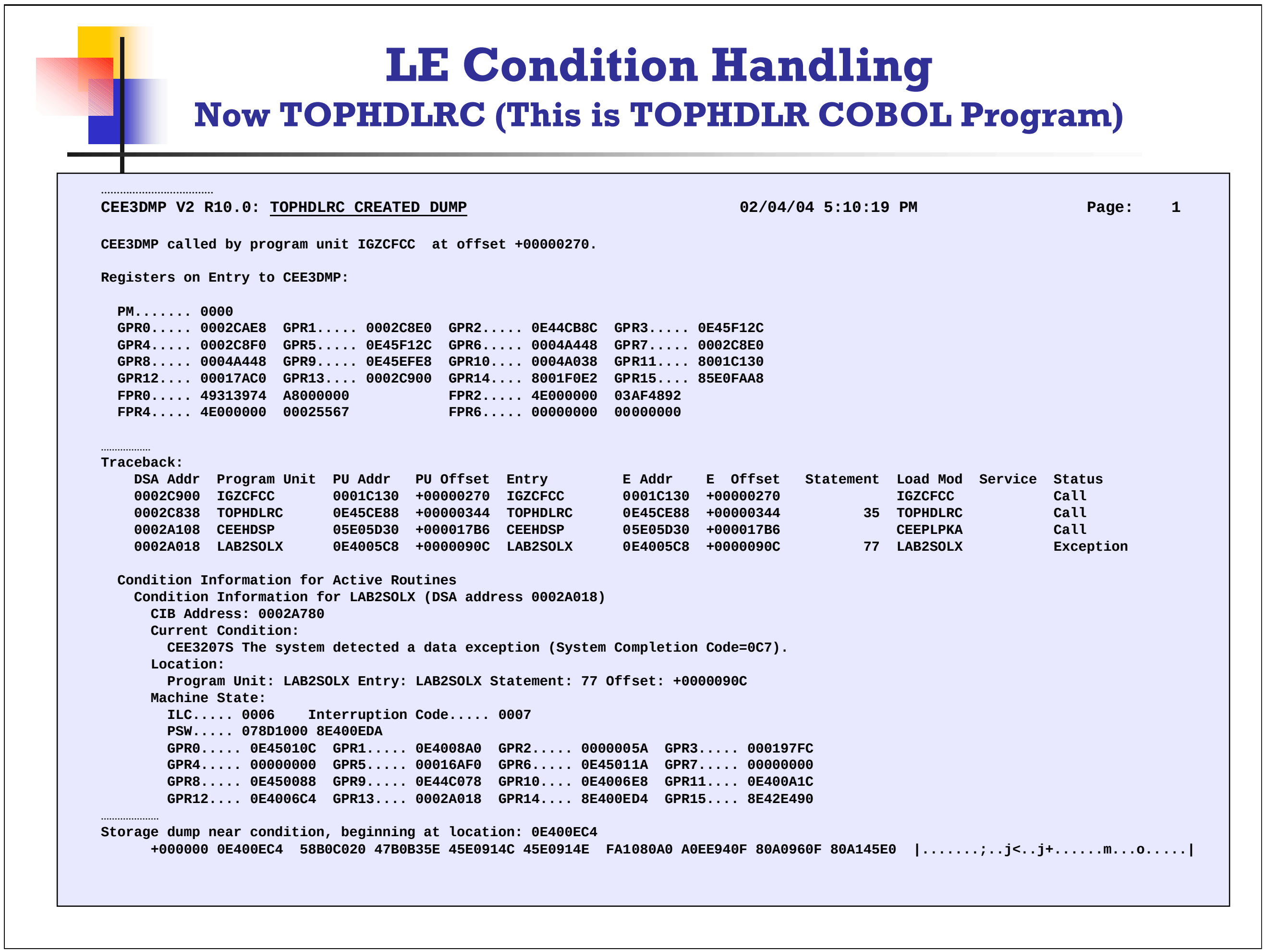


LE Condition Handling

<SNIPPIT> of dump output from the execution with LAB2 ….

```
…………………………………
Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Load Mod  Service  Status
    0002CFB0  IGZCFCC       0001C130  +00000270  IGZCFCC       0001C130  +00000270              IGZCFCC            Call
    0002CEE8  LAB2HDLR      0E458800  +00000508  LAB2HDLR      0E458800  +00000508          68  LAB2HDLR           Call
    0002A7B8  CEEHDSP       05E05D30  +000017B6  CEEHDSP       05E05D30  +000017B6              CEEPLPKA           Call
    0002A620  CEEHSGLT      05E71300  +0000005C  CEEHSGLT      05E71300  +0000005C              CEEPLPKA          Exception
    0002A108  IGZCMSG       0E4356A8  +0000038C  IGZCMSG       0E4356A8  +0000038C              IGZCPAC            Call
    0002A018  LAB2SOLX      0E4005C8  +000008A4  LAB2SOLX      0E4005C8  +000008A4          71  LAB2SOLX           Call
  Condition Information for Active Routines
    Condition Information for CEEHSGLT (DSA address 0002A620)
      CIB Address: 0002AE30
      Current Condition:
        IGZ0072S A reference modification start position value of 11 on line 000071 referenced an area outside the region
                of data item DDANO-IN.
      Location:
        Program Unit: CEEHSGLT Entry: CEEHSGLT Statement:  Offset: +0000005C
……………………….. .
Local Variables:
                 9 01 DMP-TITLE        X(80) DISP       'CEEDUMP FROM LAB2HDLR ROUTINE'                                         '
                12 01 DMP-OPTIONS      X(255) DISP      'TRACE FILE VAR STOR'
                15 01 FEEDBACK         AN-GR
                16  02 FB-SEV           S9999 COMP       +00000
                17  02 FB-MSGNO         S9999 COMP       +00000
                18  02 FB-CASE-SEV      X DISP           '?'
                19  02 FB-FAC-ID        XXX DISP         '???'
                20  02 FB-ISINFO        S9(8) COMP       +0000000000
                22 01 RESUME-CURSOR-REPOSITION
                                       AN-GR
                23  02 MOVE-TYPE-0      S9(9) COMP       +0000000000
                25  02 MOVE-TYPE-1      S9(9) COMP       +0000000001
                28 01 ERROR-INDICATOR  X DISP           'N'
                31 01 CURRENT-CONDITION
                                       AN-GR
                32  02 FBCODE           X(8) DISP        '? ?‡áIGZ'
                38  02 FILLER           XXXX DISP        '??? '
                40 01 DATA-INFO        S9(8) COMP       +0000000000
                41 01 RESULT-CODE      S9(9) COMP       +0000000020
                44 01 NEW-CONDITION    X(12) DISP       '????????????'
```

Notice the sequence of routines in the traceback sequence. The 1st program is LAB2SOLX, then a COBOL routine (IGZ…) which detects a "problem", and then the CEE routine is next, and LAB2HDLR is invoked (this is the condition handler routine that SPECIFICALLY checks for specific IGZ type errors). Notice also that the STATEMENT number is identified in the traceback. This is the last statement executed (from the compiler listing sequence number) in LAB2SOLX and LAB2HDLR, and reflects the actual last statement, NOT THE NEXT ONE, and is produced because the COBOL programs were compiled with TEST(NONE,SYM). The SYM produces the symbol table that creates this statement number, and, in addition, produces the LOCAL VARIABLES portion of the dump with the data names and data field contents.

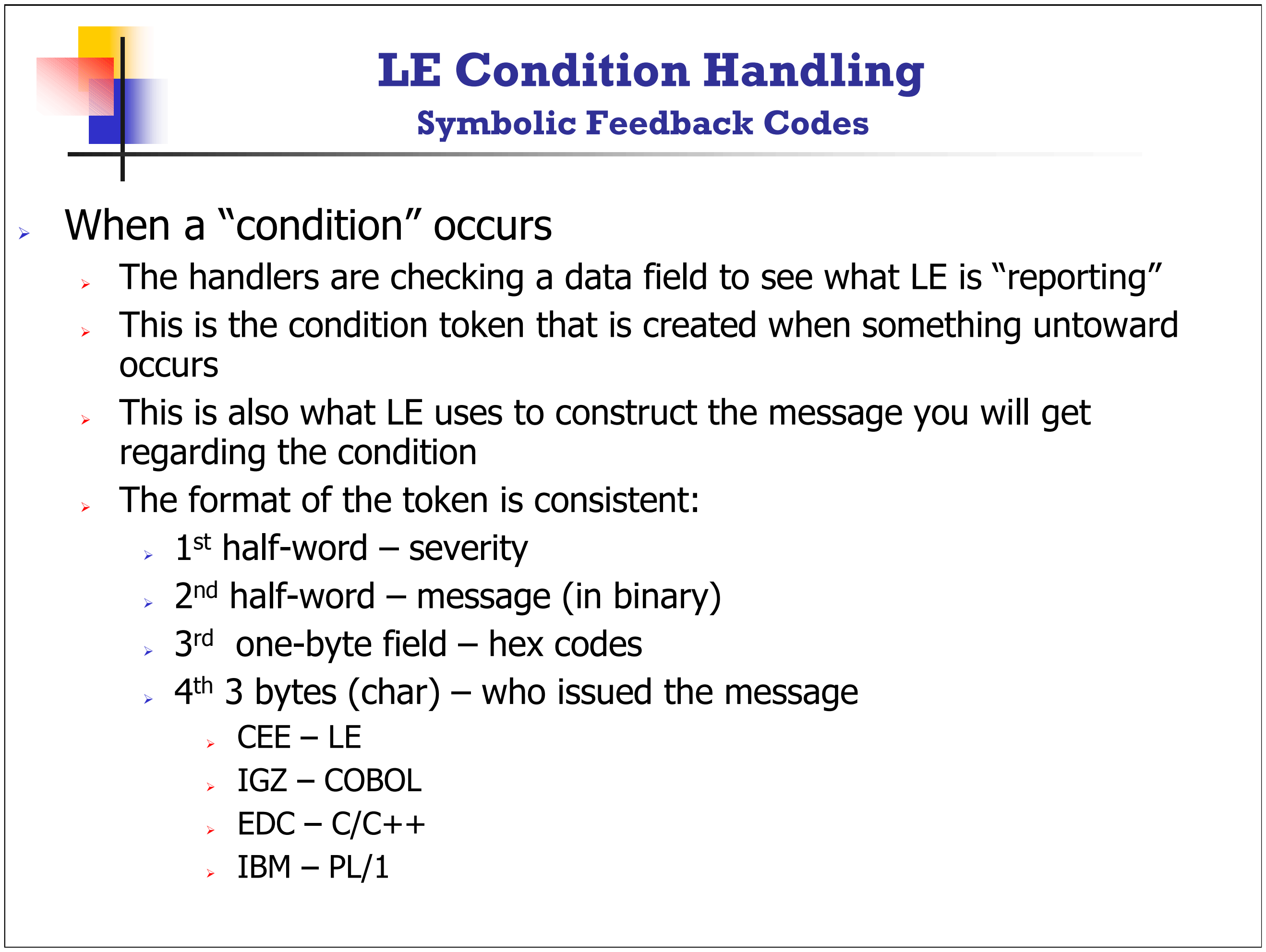


LE Condition Handling

Now TOPHDLRC (This is TOPHDLR COBOL Program)

```
CEE3DMP V2 R10.0: TOPHDLRC CREATED DUMP                          02/04/04 5:10:19 PM                  Page:    1

CEE3DMP called by program unit IGZCFCC  at offset +00000270.

Registers on Entry to CEE3DMP:

  PM....... 0000
  GPR0..... 0002CAE8  GPR1..... 0002C8E0  GPR2..... 0E44CB8C  GPR3..... 0E45F12C
  GPR4..... 0002C8F0  GPR5..... 0E45F12C  GPR6..... 0004A448  GPR7..... 0002C8E0
  GPR8..... 0004A448  GPR9..... 0E45EFE8  GPR10.... 0004A038  GPR11.... 8001C130
  GPR12.... 00017AC0  GPR13.... 0002C900  GPR14.... 8001F0E2  GPR15.... 85E0FAA8
  FPR0..... 49313974  A8000000            FPR2..... 4E000000  03AF4892
  FPR4..... 4E000000  00025567            FPR6..... 00000000  00000000


Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry          E Addr    E  Offset   Statement  Load Mod  Service  Status
    0002C900  IGZCFCC       0001C130  +00000270  IGZCFCC        0001C130  +00000270              IGZCFCC            Call
    0002C838  TOPHDLRC      0E45CE88  +00000344  TOPHDLRC       0E45CE88  +00000344          35  TOPHDLRC           Call
    0002A108  CEEHDSP       05E05D30  +000017B6  CEEHDSP        05E05D30  +000017B6              CEEPLPKA           Call
    0002A018  LAB2SOLX      0E4005C8  +0000090C  LAB2SOLX       0E4005C8  +0000090C          77  LAB2SOLX           Exception

  Condition Information for Active Routines
    Condition Information for LAB2SOLX (DSA address 0002A018)
      CIB Address: 0002A780
      Current Condition:
        CEE3207S The system detected a data exception (System Completion Code=0C7).
      Location:
        Program Unit: LAB2SOLX Entry: LAB2SOLX Statement: 77 Offset: +0000090C
      Machine State:
        ILC..... 0006    Interruption Code..... 0007
        PSW..... 078D1000 8E400EDA
        GPR0..... 0E45010C  GPR1..... 0E4008A0  GPR2..... 0000005A  GPR3..... 000197FC
        GPR4..... 00000000  GPR5..... 00016AF0  GPR6..... 0E45011A  GPR7..... 00000000
        GPR8..... 0E450088  GPR9..... 0E44C078  GPR10.... 0E4006E8  GPR11.... 0E400A1C
        GPR12.... 0E4006C4  GPR13.... 0002A018  GPR14.... 8E400ED4  GPR15.... 8E42E490

Storage dump near condition, beginning at location: 0E400EC4
      +000000 0E400EC4  58B0C020 47B0B35E 45E0914C 45E0914E  FA1080A0 A0EE940F 80A0960F 80A145E0  |.......;..j<..j+......m...o.....|
```

This dump is produced because of the ADD +1 to DUMP-IT in the program, a purposeful setup to create a SOC7 abend AFTER the original COBOL reference-modification outside range of data item error. This program was “registered” in LAB2SOLX after the LAB2HDLR condition handler was registered. In this example it does not really matter which program is registered first, this was merely to show that you can get multiple handlers involved, and depending on the action the handler takes, in this example, RESUME, both handlers see action!

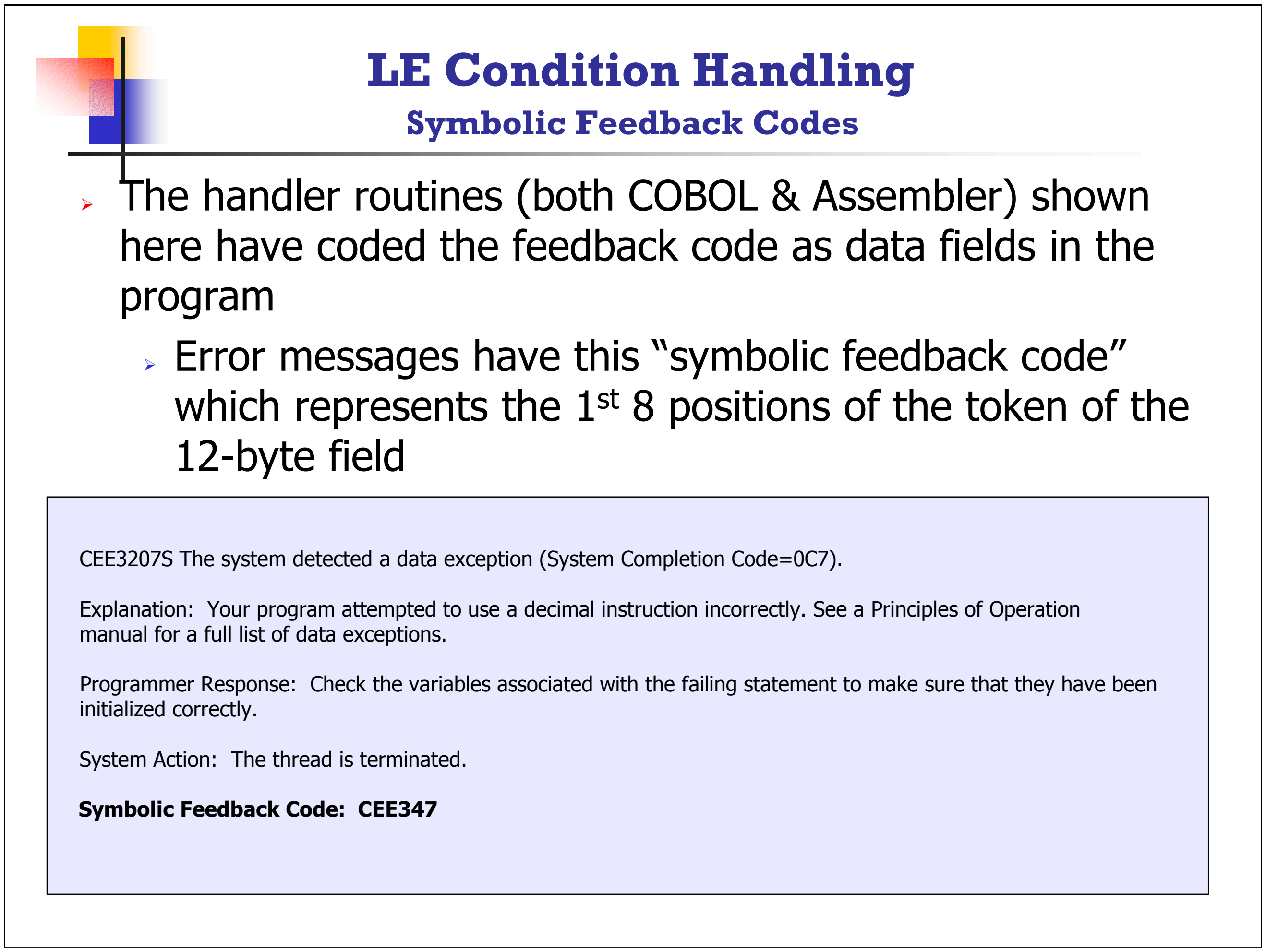


LE Condition Handling

Symbolic Feedback Codes

- When a “condition” occurs
  - The handlers are checking a data field to see what LE is “reporting”
  - This is the condition token that is created when something untoward occurs
  - This is also what LE uses to construct the message you will get regarding the condition
  - The format of the token is consistent:
    - 1st half-word – severity
    - 2nd half-word – message (in binary)
    - 3rd one-byte field – hex codes
    - 4th 3 bytes (char) – who issued the message
      - CEE – LE
      - IGZ – COBOL
      - EDC – C/C++
      - IBM – PL/1

The format of the condition token used for condition handling is exactly the same as the feedback token from any call to any of the LE callable services, and the use is the same. The 1st half-word is the severity (like a status back from the CALL to a CEE… routine), etc.



LE Condition Handling

Symbolic Feedback Codes

- The handler routines (both COBOL & Assembler) shown here have coded the feedback code as data fields in the program
  - Error messages have this "symbolic feedback code" which represents the 1st 8 positions of the token of the 12-byte field

CEE3207S The system detected a data exception (System Completion Code=0C7).

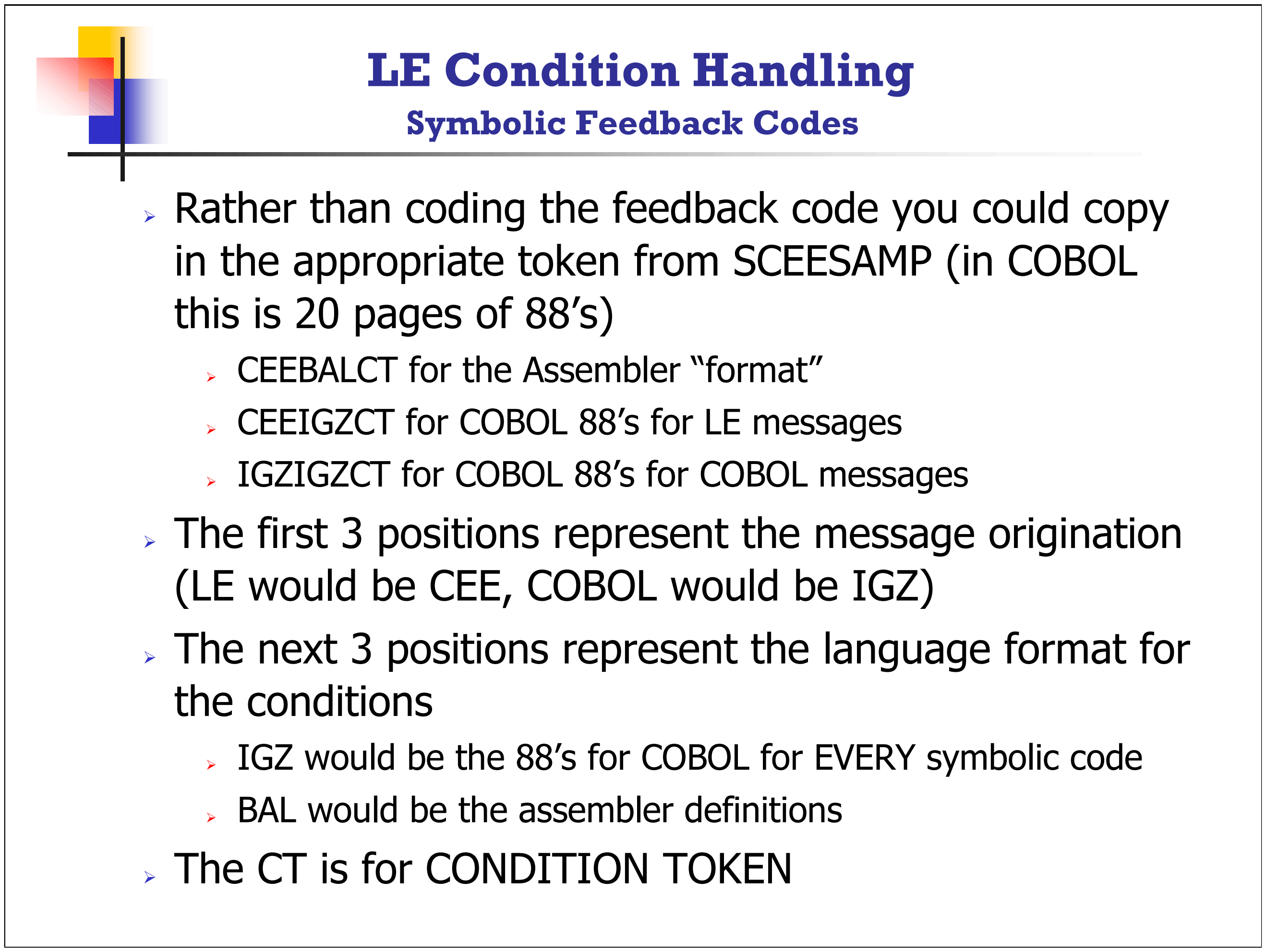
Explanation: Your program attempted to use a decimal instruction incorrectly. See a Principles of Operation manual for a full list of data exceptions.

Programmer Response: Check the variables associated with the failing statement to make sure that they have been initialized correctly.

System Action: The thread is terminated.

Symbolic Feedback Code: CEE347

This message information is from the LE Run-time messages manual. This information is the "tie that binds" that allow you identify the symbolic feedback code which can be used to uncover the actual information in the 12-byte condition token.

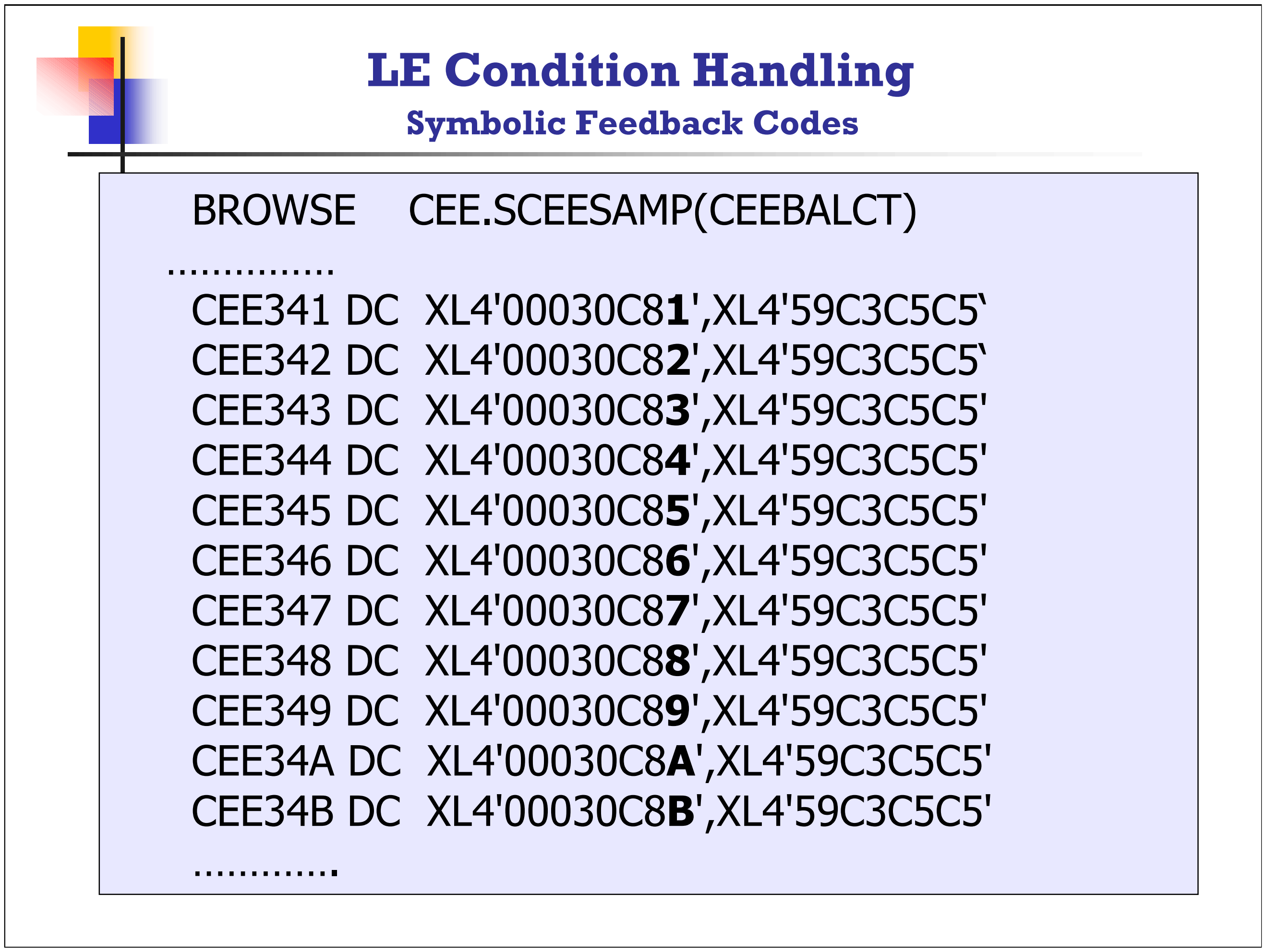


LE Condition Handling

Symbolic Feedback Codes

- Rather than coding the feedback code you could copy in the appropriate token from SCEESAMP (in COBOL this is 20 pages of 88's)
  - CEEBALCT for the Assembler "format"
  - CEEIGZCT for COBOL 88's for LE messages
  - IGZIGZCT for COBOL 88's for COBOL messages
- The first 3 positions represent the message origination (LE would be CEE, COBOL would be IGZ)
- The next 3 positions represent the language format for the conditions
  - IGZ would be the 88's for COBOL for EVERY symbolic code
  - BAL would be the assembler definitions
- The CT is for CONDITION TOKEN

The MESSAGE number is documented in the LE run-time messages manual. Each message has an associated SYMBOLIC FEEDBACK associated with it. If you examine the members in SCEESAMP for the appropriate language, IGZCEECT for COBOL symbolic feedbacks for LE originating message, IGZIGZCT for condition token definitions for COBOL program format, COBOL IGZ messages, etc. The symbolic feedback code files have the file names as: xxxyyyCT, where XXX indicates the facility id for WHO IS ISSUING the message, and yyy is the handler language format, such as BAL for assembler, IGZ for COBOL. So, if you want to find the assembler formats for symbolic feedback codes, you would go to SCEESAMP, locate IGZBALCT for the assembler format of COBOL error feedbacks, or CEEIGZCT, as another example, for COBOL format (88-level) of the Language Environment feedback codes. The detail explanation of the symbolic feedback code is explained in great detail in the LE Programming Guide.

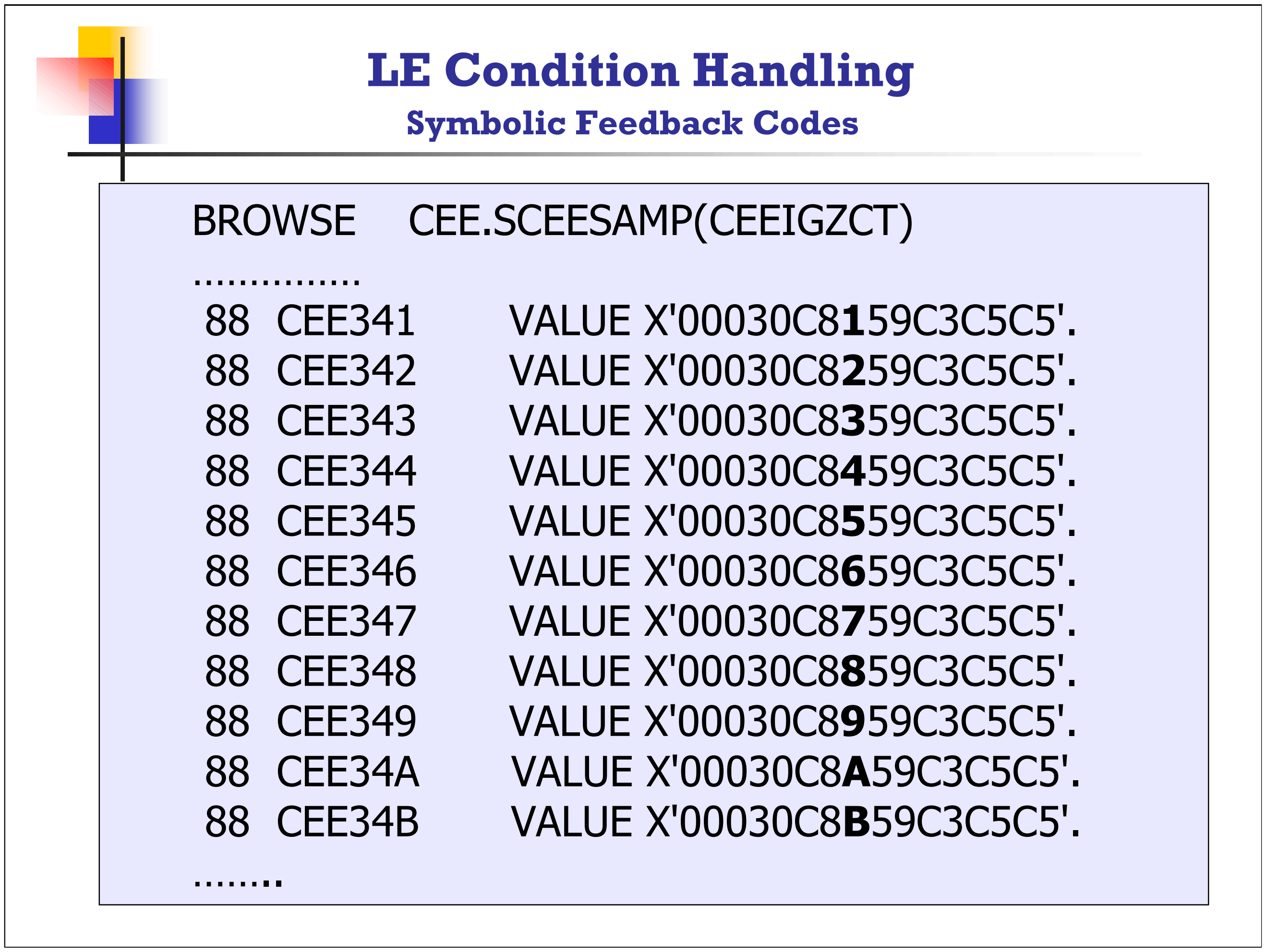


LE Condition Handling

Symbolic Feedback Codes

```
  BROWSE    CEE.SCEESAMP(CEEBALCT)
...............
  CEE341 DC  XL4'00030C81',XL4'59C3C5C5`
  CEE342 DC  XL4'00030C82',XL4'59C3C5C5`
  CEE343 DC  XL4'00030C83',XL4'59C3C5C5'
  CEE344 DC  XL4'00030C84',XL4'59C3C5C5'
  CEE345 DC  XL4'00030C85',XL4'59C3C5C5'
  CEE346 DC  XL4'00030C86',XL4'59C3C5C5'
  CEE347 DC  XL4'00030C87',XL4'59C3C5C5'
  CEE348 DC  XL4'00030C88',XL4'59C3C5C5'
  CEE349 DC  XL4'00030C89',XL4'59C3C5C5'
  CEE34A DC  XL4'00030C8A',XL4'59C3C5C5'
  CEE34B DC  XL4'00030C8B',XL4'59C3C5C5'
  ………….
```

This is a sample copied from the assembler format of the LE messages for the typical SOC1 through SOCB program abend errors. The symbolic feedback code CEE347 is the one for a data exception, the actual "message" is CEE3207S…and the 7 is consistent in the message and the feedback and it is, the 7 from the SOC7 you know and love!

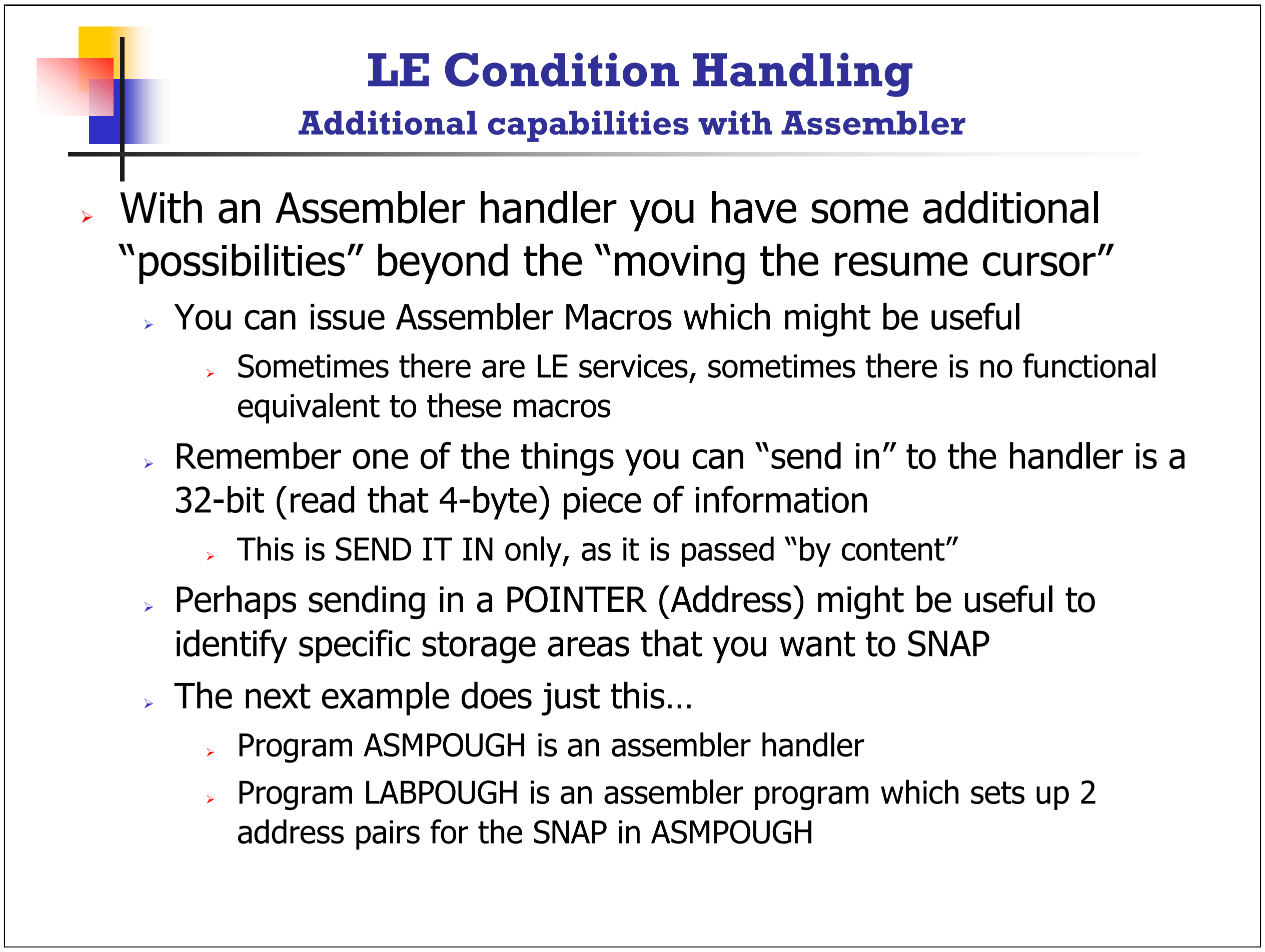


LE Condition Handling

Symbolic Feedback Codes

```
BROWSE    CEE.SCEESAMP(CEEIGZCT)
...............
 88  CEE341       VALUE X'00030C8159C3C5C5'.
 88  CEE342       VALUE X'00030C8259C3C5C5'.
 88  CEE343       VALUE X'00030C8359C3C5C5'.
 88  CEE344       VALUE X'00030C8459C3C5C5'.
 88  CEE345       VALUE X'00030C8559C3C5C5'.
 88  CEE346       VALUE X'00030C8659C3C5C5'.
 88  CEE347       VALUE X'00030C8759C3C5C5'.
 88  CEE348       VALUE X'00030C8859C3C5C5'.
 88  CEE349       VALUE X'00030C8959C3C5C5'.
 88  CEE34A       VALUE X'00030C8A59C3C5C5'.
 88  CEE34B       VALUE X'00030C8B59C3C5C5'.
........
```

This is a sample for the COBOL formats for the LE CEE messages for SOC1 – SOCB.



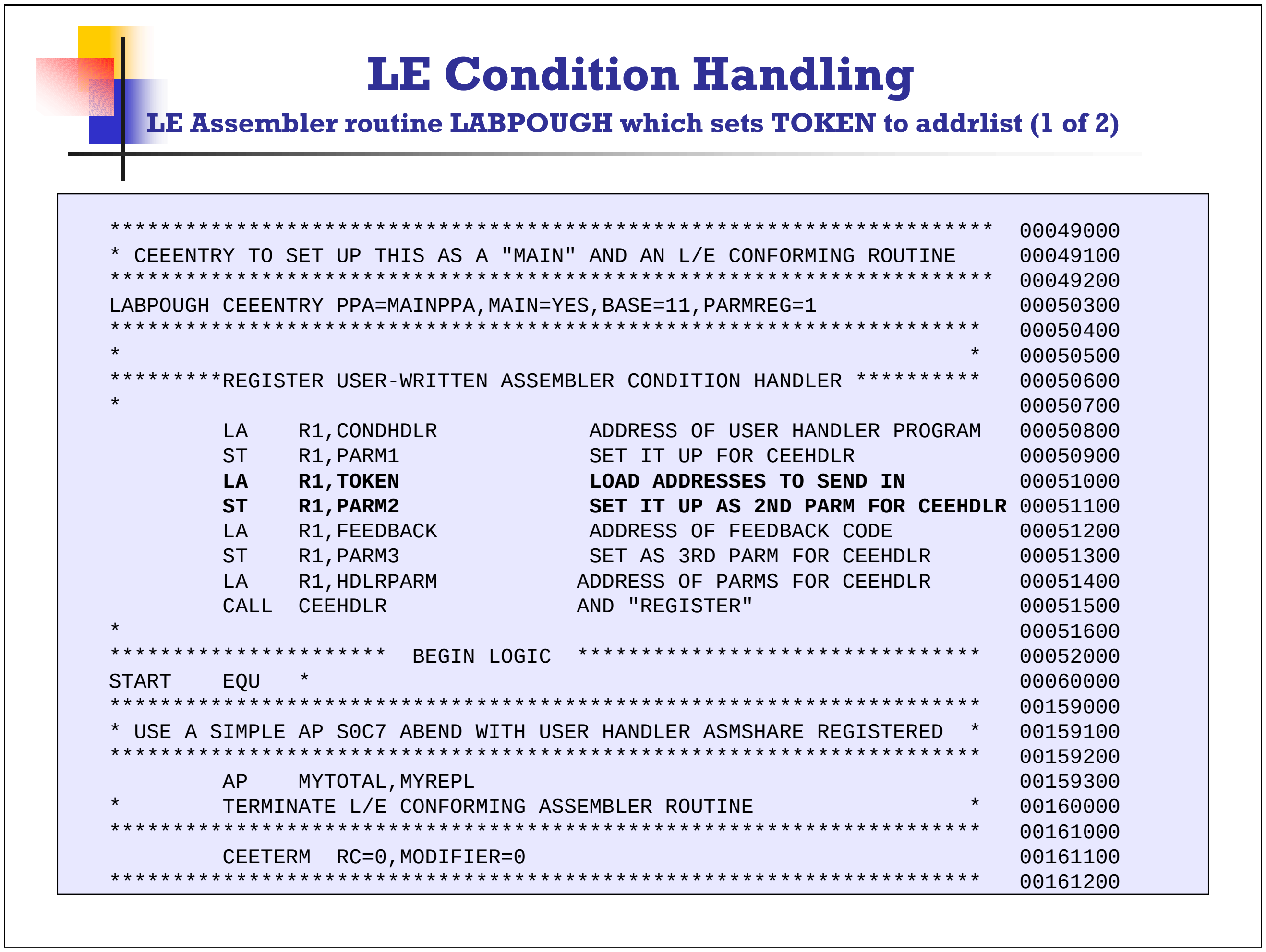
LE Condition Handling

Additional capabilities with Assembler

- With an Assembler handler you have some additional "possibilities" beyond the "moving the resume cursor"
  - You can issue Assembler Macros which might be useful
    - Sometimes there are LE services, sometimes there is no functional equivalent to these macros
  - Remember one of the things you can "send in" to the handler is a 32-bit (read that 4-byte) piece of information
    - This is SEND IT IN only, as it is passed "by content"
  - Perhaps sending in a POINTER (Address) might be useful to identify specific storage areas that you want to SNAP
  - The next example does just this…
    - Program ASMPOUGH is an assembler handler
    - Program LABPOUGH is an assembler program which sets up 2 address pairs for the SNAP in ASMPOUGH

The LE Programming Reference has an excellent chart with a list of Assembler Macros, comparable LE services, if available, and whether the Assembler macro can be used in an LE-conforming assembler program. Here is an example from this table:

SNAP	Call CEE3DMP.	This service can be used.
STIMER(2) .	No equivalent Language Environment function.	This service can be used.
TIME .	Call Language Environment date and time services.	This service can be used.
SVC LINK .	No equivalent Language Environment function	This service can be used. For compatibility, Language Environment supports the LINK boundary crossing and treats it as a new enclave.
WAIT/POST /EVENTS(3) .	No equivalent Language Environment function.	Host services can be used.
WTO .	Call CEEMOUT. This writes to the error log or the terminal.	Host services can be used.
XCTL .	No equivalent Language Environment function.	Host services can, but should not, be used.

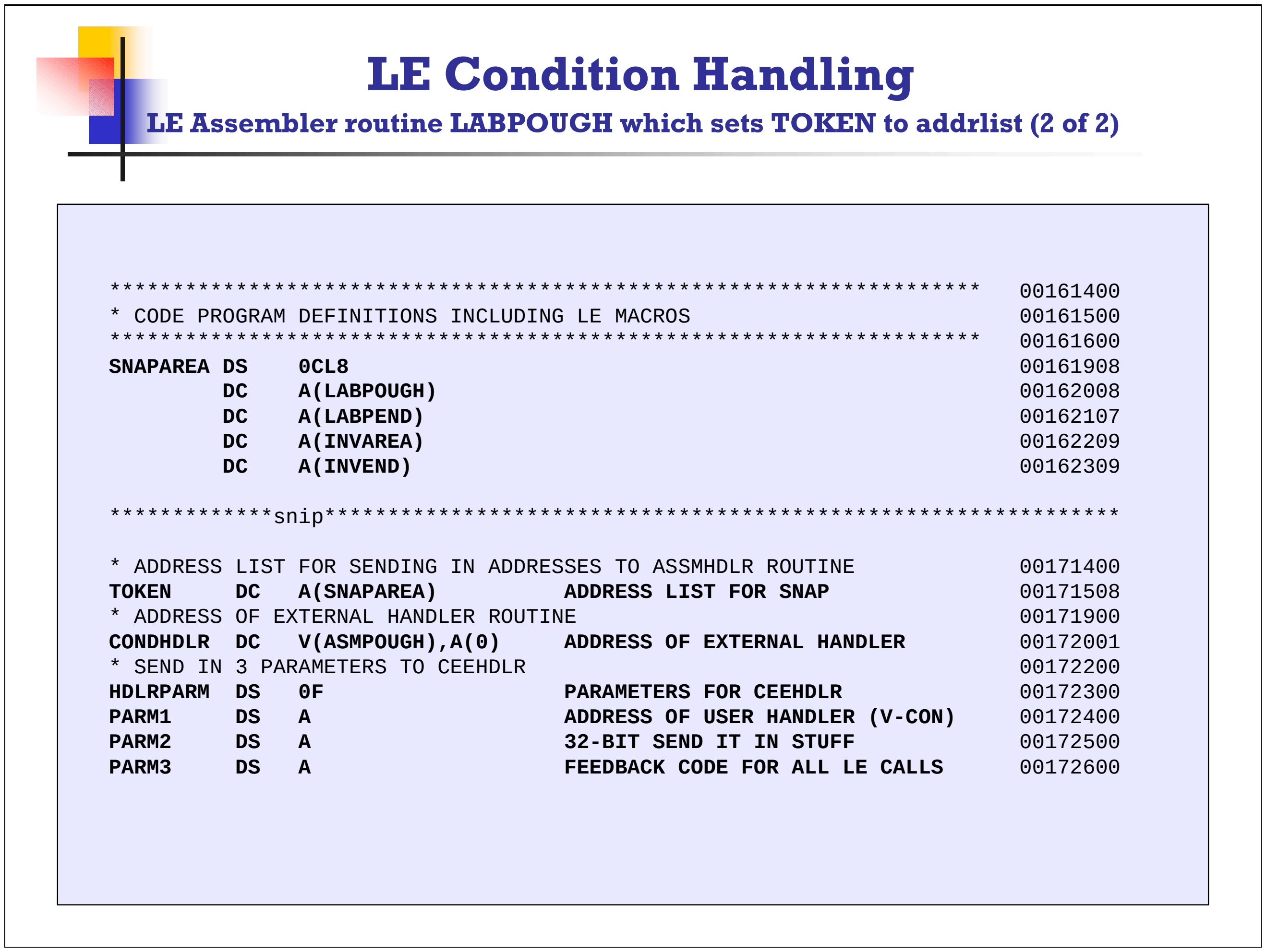


LE Condition Handling

LE Assembler routine LABPOUGH which sets TOKEN to addrlist (1 of 2)

```
**********************************************************************
* CEEENTRY TO SET UP THIS AS A "MAIN" AND AN L/E CONFORMING ROUTINE
**********************************************************************
LABPOUGH CEEENTRY PPA=MAINPPA,MAIN=YES,BASE=11,PARMREG=1
*********************************************************************
*                                                                   *
*********REGISTER USER-WRITTEN ASSEMBLER CONDITION HANDLER **********
*
         LA    R1,CONDHDLR            ADDRESS OF USER HANDLER PROGRAM
         ST    R1,PARM1               SET IT UP FOR CEEHDLR
         LA    R1,TOKEN               LOAD ADDRESSES TO SEND IN
         ST    R1,PARM2               SET IT UP AS 2ND PARM FOR CEEHDLR
         LA    R1,FEEDBACK            ADDRESS OF FEEDBACK CODE
         ST    R1,PARM3               SET AS 3RD PARM FOR CEEHDLR
         LA    R1,HDLRPARM           ADDRESS OF PARMS FOR CEEHDLR
         CALL  CEEHDLR               AND "REGISTER"
*
**********************  BEGIN LOGIC  ********************************
START    EQU   *
*********************************************************************
* USE A SIMPLE AP S0C7 ABEND WITH USER HANDLER ASMSHARE REGISTERED  *
*********************************************************************
         AP    MYTOTAL,MYREPL
*        TERMINATE L/E CONFORMING ASSEMBLER ROUTINE                 *
*********************************************************************
         CEETERM  RC=0,MODIFIER=0
*********************************************************************
```

This is a portion of the Assembler program that is setting up addresses to be specifically “snap’d” in the condition handler routine. When registering a user condition handler with the call to CEEHDLR, the 2n’d parameter can be a pointer (address), and in this example the address points to 2 address pairs so that the condition handler can issue the SNAP macro to capture specific identified areas. This technique obviously requires coordination between the application and the condition handler, and can ONLY be implemented by using the CEEHDLR call to register the user handler, not via the PARM=‘/USRHDLR’, where you do not have the opportunity to identify the 2nd parameter for CEEHDLR.

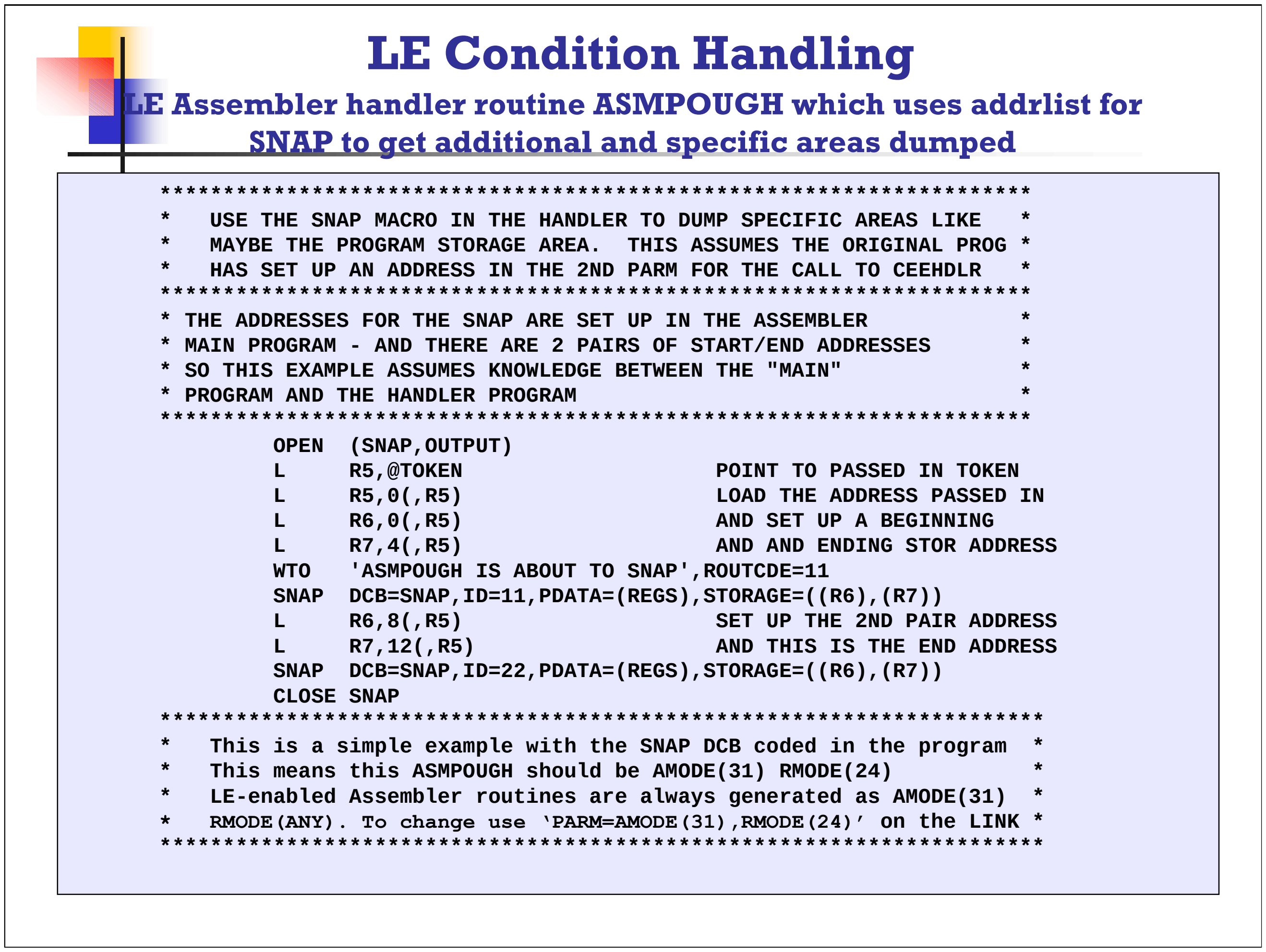


LE Condition Handling

LE Assembler routine LABPOUGH which sets TOKEN to addrlist (2 of 2)

```
*********************************************************************
* CODE PROGRAM DEFINITIONS INCLUDING LE MACROS
*********************************************************************
SNAPAREA DS    0CL8
         DC    A(LABPOUGH)
         DC    A(LABPEND)
         DC    A(INVAREA)
         DC    A(INVEND)

*************snip******************************************************************

* ADDRESS LIST FOR SENDING IN ADDRESSES TO ASSMHDLR ROUTINE
TOKEN     DC   A(SNAPAREA)          ADDRESS LIST FOR SNAP
* ADDRESS OF EXTERNAL HANDLER ROUTINE
CONDHDLR  DC   V(ASMPOUGH),A(0)     ADDRESS OF EXTERNAL HANDLER
* SEND IN 3 PARAMETERS TO CEEHDLR
HDLRPARM  DS   0F                   PARAMETERS FOR CEEHDLR
PARM1     DS   A                    ADDRESS OF USER HANDLER (V-CON)
PARM2     DS   A                    32-BIT SEND IT IN STUFF
PARM3     DS   A                    FEEDBACK CODE FOR ALL LE CALLS
```

The SNAPAREA sets up 2 “address pairs” that can be sent into the condition handler (Assembler) so that a SNAP can dump specific identified areas. Using CEEDUMP to produce dump information is useful, but sometimes you may wish to identify some user specific areas that need to be available, and this technique might be preferable to producing a complete region dump. The SNAPAREA address will be passed into the condition handler as the 2nd parameter, and the condition handler “understands” that these address pairs are set up in the originating application, and can then SNAP them.

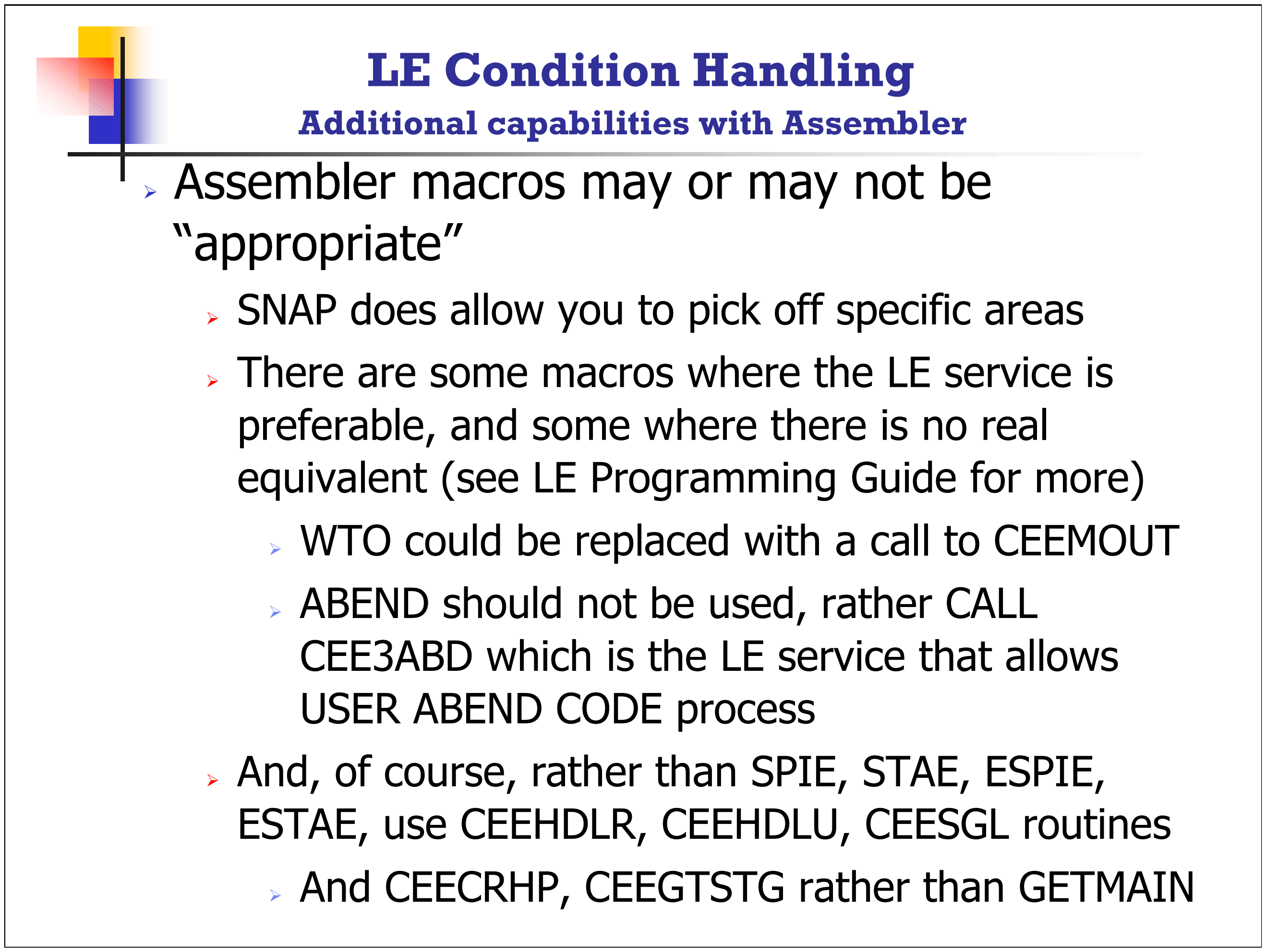


LE Condition Handling

LE Assembler handler routine ASMPOUGH which uses addrlist for SNAP to get additional and specific areas dumped

```
*********************************************************************
*   USE THE SNAP MACRO IN THE HANDLER TO DUMP SPECIFIC AREAS LIKE   *
*   MAYBE THE PROGRAM STORAGE AREA.  THIS ASSUMES THE ORIGINAL PROG *
*   HAS SET UP AN ADDRESS IN THE 2ND PARM FOR THE CALL TO CEEHDLR   *
*********************************************************************
* THE ADDRESSES FOR THE SNAP ARE SET UP IN THE ASSEMBLER            *
* MAIN PROGRAM - AND THERE ARE 2 PAIRS OF START/END ADDRESSES       *
* SO THIS EXAMPLE ASSUMES KNOWLEDGE BETWEEN THE "MAIN"               *
* PROGRAM AND THE HANDLER PROGRAM                                   *
*********************************************************************
         OPEN  (SNAP,OUTPUT)
         L     R5,@TOKEN                    POINT TO PASSED IN TOKEN
         L     R5,0(,R5)                    LOAD THE ADDRESS PASSED IN
         L     R6,0(,R5)                    AND SET UP A BEGINNING
         L     R7,4(,R5)                    AND AND ENDING STOR ADDRESS
         WTO   'ASMPOUGH IS ABOUT TO SNAP',ROUTCDE=11
         SNAP  DCB=SNAP,ID=11,PDATA=(REGS),STORAGE=((R6),(R7))
         L     R6,8(,R5)                    SET UP THE 2ND PAIR ADDRESS
         L     R7,12(,R5)                   AND THIS IS THE END ADDRESS
         SNAP  DCB=SNAP,ID=22,PDATA=(REGS),STORAGE=((R6),(R7))
         CLOSE SNAP
**********************************************************************
*   This is a simple example with the SNAP DCB coded in the program  *
*   This means this ASMPOUGH should be AMODE(31) RMODE(24)           *
*   LE-enabled Assembler routines are always generated as AMODE(31)  *
*   RMODE(ANY). To change use 'PARM=AMODE(31),RMODE(24)' on the LINK *
**********************************************************************
```

This is a portion of the condition handler routine ASMPOUGH, which expects that a pair of addresses have been set up in the main program (or the program which “registers” the user condition handler via the call to CEEHDLR). Remember also that LE conforming assembler programs are automatically set to be AMODE(31), RMODE(ANY), so if you have a DCB in your assembler application you need to FORCE the AMODE and RMODE via the PARM at LINK or BINDER time. You CANNOT use the AMODE and RMODE directive, you MUST use the PARM technique, but by making the routine AMOD(31), RMODE(24) this assembler handler can deal with either AMODE(31) or AMODE(24) programs! The handler routine MUST BE SEPARATELY LINKEDT from the application programs, and will only be loaded IF YOU ENCOUNTER A CONDITION!

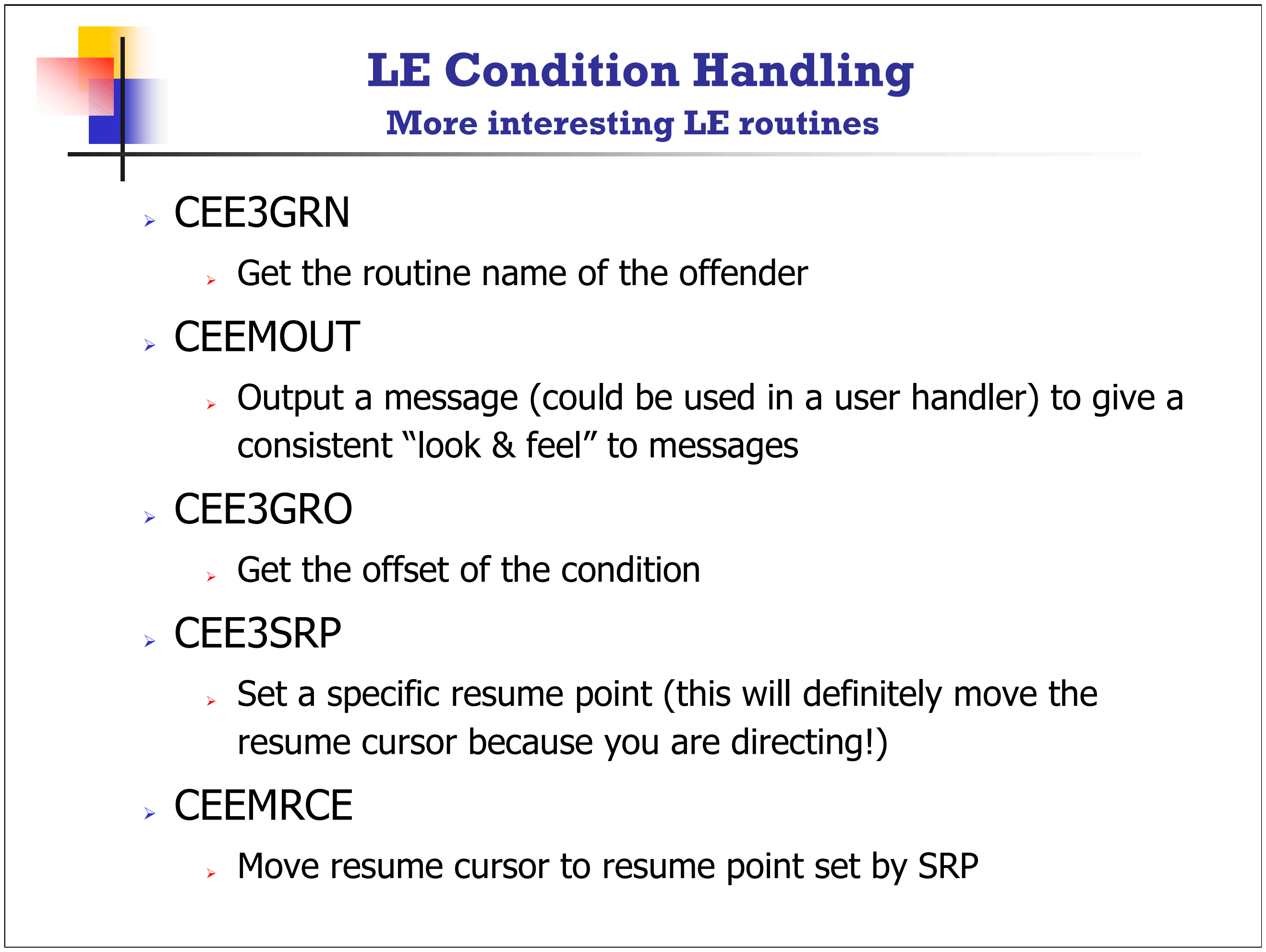


LE Condition Handling

Additional capabilities with Assembler

- Assembler macros may or may not be "appropriate"
  - SNAP does allow you to pick off specific areas
  - There are some macros where the LE service is preferable, and some where there is no real equivalent (see LE Programming Guide for more)
    - WTO could be replaced with a call to CEEMOUT
    - ABEND should not be used, rather CALL CEE3ABD which is the LE service that allows USER ABEND CODE process
  - And, of course, rather than SPIE, STAE, ESPIE, ESTAE, use CEEHDLR, CEEHDLU, CEESGL routines
    - And CEECRHP, CEEGTSTG rather than GETMAIN

Although the LE manuals indicate that CEEDUMP is a "replacement" for SNAP, clearly there are some capabilities by using SNAP that are not available when using CEEDUMP. In other cases, the LE equivalent may provide replacement AND MORE.

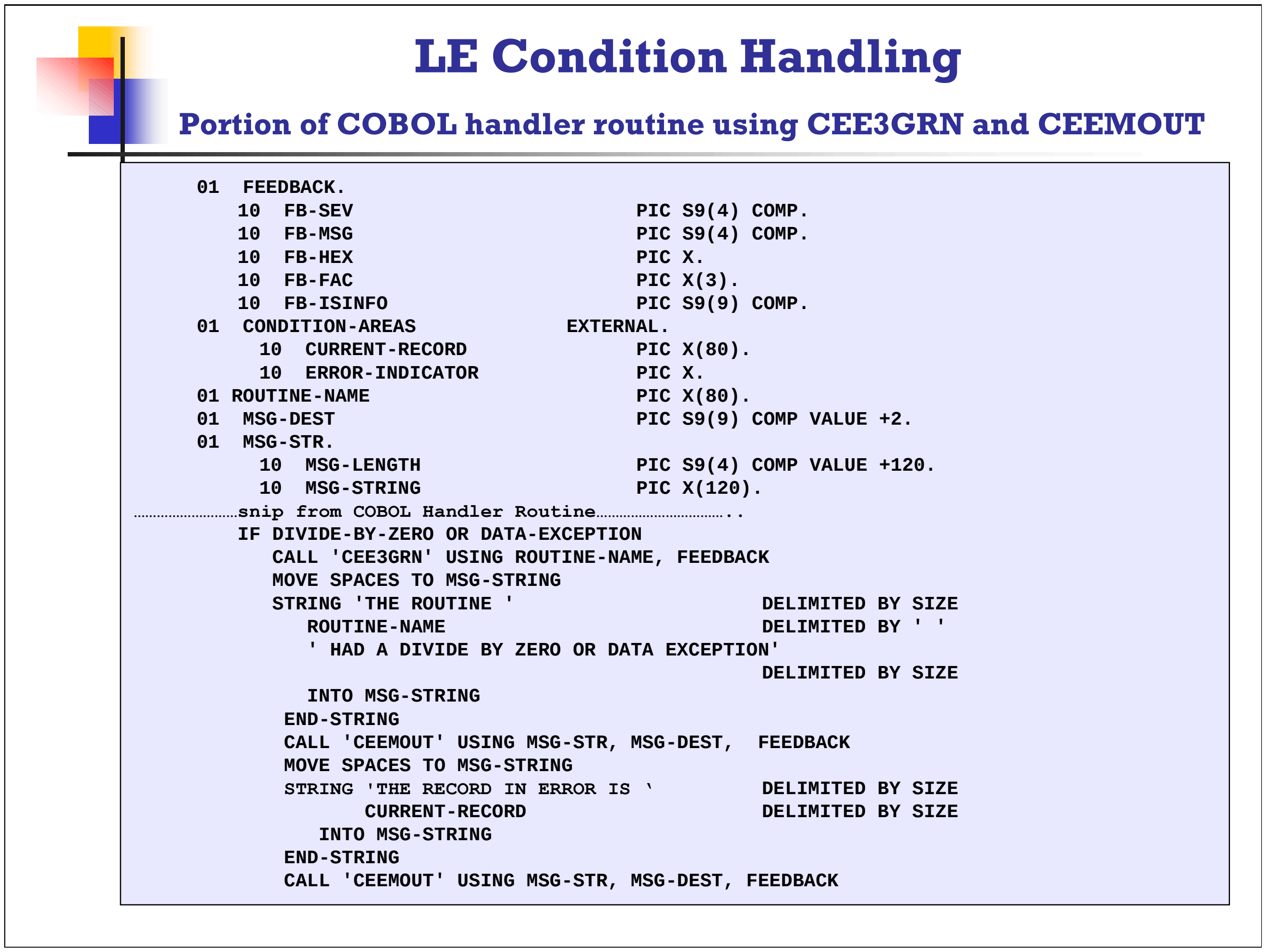


LE Condition Handling

More interesting LE routines

- CEE3GRN
  - Get the routine name of the offender
- CEEMOUT
  - Output a message (could be used in a user handler) to give a consistent "look & feel" to messages
- CEE3GRO
  - Get the offset of the condition
- CEE3SRP
  - Set a specific resume point (this will definitely move the resume cursor because you are directing!)
- CEEMRCE
  - Move resume cursor to resume point set by SRP

Samples of using these routines are shown in the LE Programming Guide. Typically using CEE3DUMP will give you the routine name of the offender, but there may be situations where all you want to do is write out a message (CEEMOUT) with the name of the offending routine, and perhaps also by using EXTERNAL data areas in COBOL, you could also "message out" the appropriate data information rather than using CEE3DMP, it is a CHOICE!

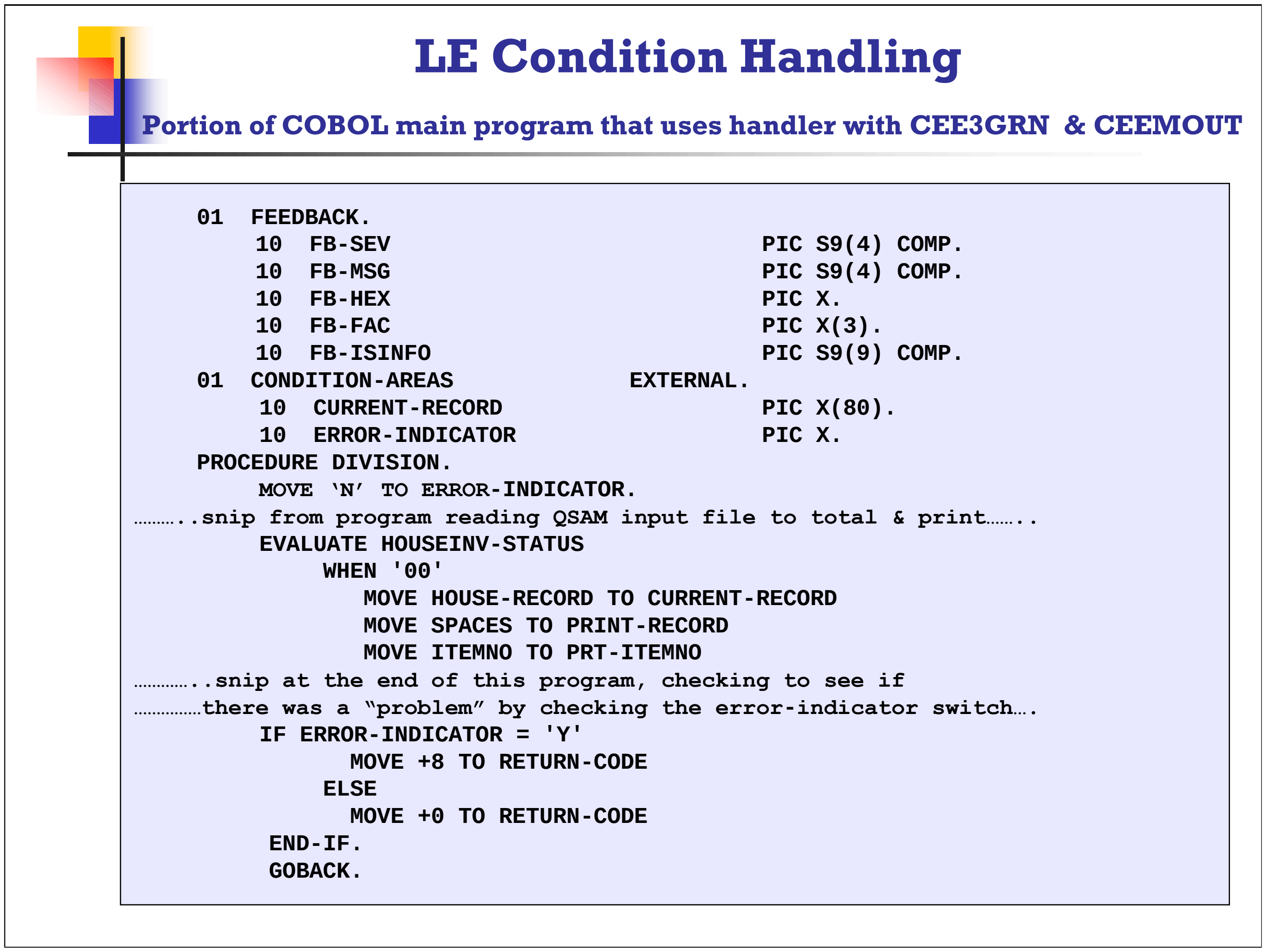


LE Condition Handling

Portion of COBOL handler routine using CEE3GRN and CEEMOUT

```
     01  FEEDBACK.
         10  FB-SEV                        PIC S9(4) COMP.
         10  FB-MSG                        PIC S9(4) COMP.
         10  FB-HEX                        PIC X.
         10  FB-FAC                        PIC X(3).
         10  FB-ISINFO                     PIC S9(9) COMP.
     01  CONDITION-AREAS             EXTERNAL.
           10  CURRENT-RECORD              PIC X(80).
           10  ERROR-INDICATOR             PIC X.
     01 ROUTINE-NAME                       PIC X(80).
     01  MSG-DEST                          PIC S9(9) COMP VALUE +2.
     01  MSG-STR.
           10  MSG-LENGTH                  PIC S9(4) COMP VALUE +120.
           10  MSG-STRING                  PIC X(120).
………………………snip from COBOL Handler Routine…………………………..
         IF DIVIDE-BY-ZERO OR DATA-EXCEPTION
            CALL 'CEE3GRN' USING ROUTINE-NAME, FEEDBACK
            MOVE SPACES TO MSG-STRING
            STRING 'THE ROUTINE '                     DELIMITED BY SIZE
               ROUTINE-NAME                           DELIMITED BY ' '
               ' HAD A DIVIDE BY ZERO OR DATA EXCEPTION'
                                                      DELIMITED BY SIZE
               INTO MSG-STRING
             END-STRING
             CALL 'CEEMOUT' USING MSG-STR, MSG-DEST,  FEEDBACK
             MOVE SPACES TO MSG-STRING
             STRING 'THE RECORD IN ERROR IS `         DELIMITED BY SIZE
                    CURRENT-RECORD                    DELIMITED BY SIZE
                INTO MSG-STRING
             END-STRING
             CALL 'CEEMOUT' USING MSG-STR, MSG-DEST, FEEDBACK
```

This is a code snip that shows using CEE3GRN in a user condition handler to get the name of the “offending routine” and write out the routine-name. In addition, this example shows an 01 data area ‘CONDITION-AREAS’ that is cross-coded in the COBOL application program and in the condition handler, so that as a record is processed it can be available for the condition handler to message out with a call to CEEMOUT. Obviously this type of condition handler requires some coordination between the application program and the condition handler.

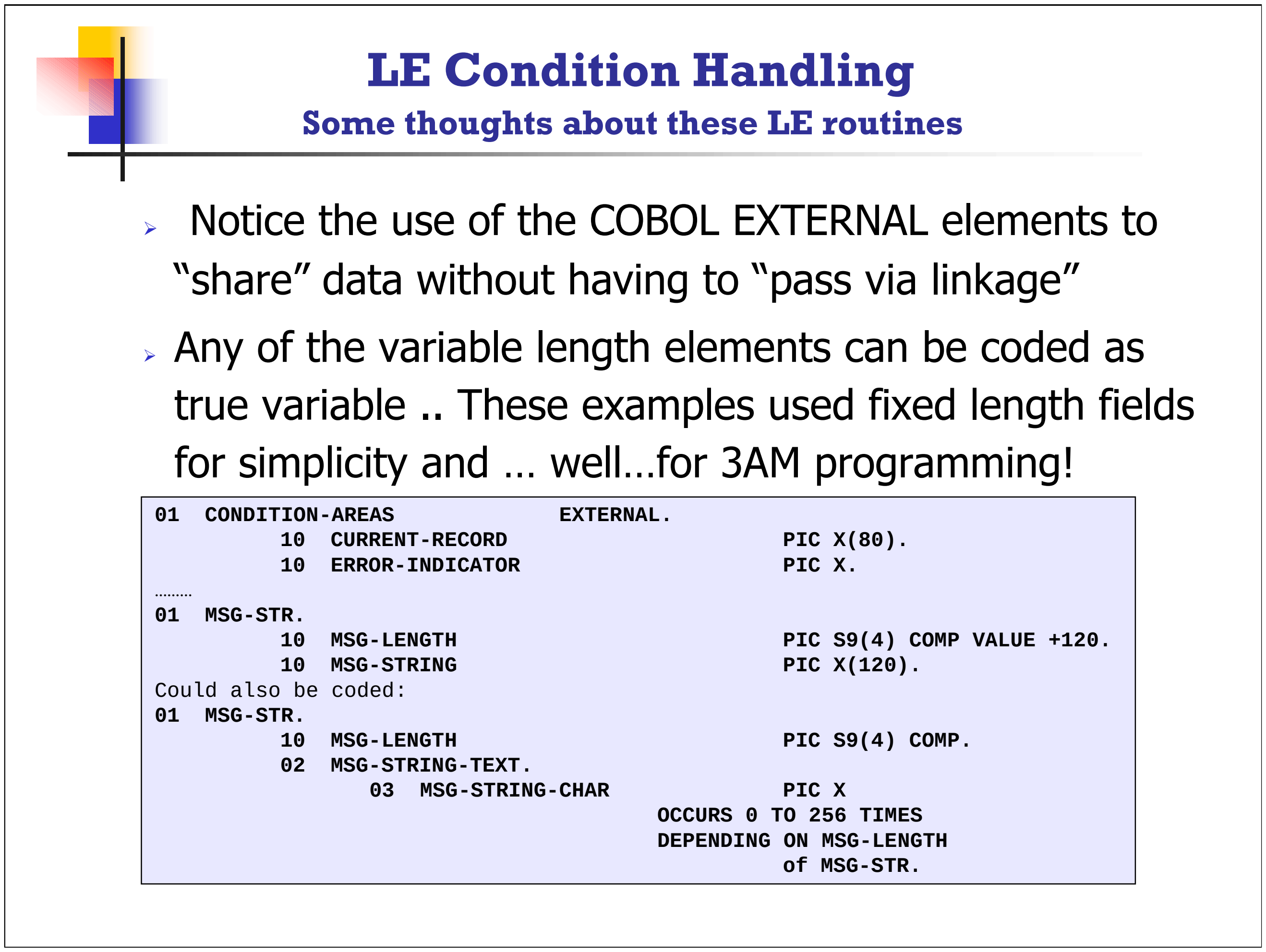


LE Condition Handling

Portion of COBOL main program that uses handler with CEE3GRN & CEEMOUT

```
     01  FEEDBACK.
         10  FB-SEV                         PIC S9(4) COMP.
         10  FB-MSG                         PIC S9(4) COMP.
         10  FB-HEX                         PIC X.
         10  FB-FAC                         PIC X(3).
         10  FB-ISINFO                      PIC S9(9) COMP.
     01  CONDITION-AREAS             EXTERNAL.
         10  CURRENT-RECORD                 PIC X(80).
         10  ERROR-INDICATOR                PIC X.
     PROCEDURE DIVISION.
          MOVE 'N' TO ERROR-INDICATOR.
……….snip from program reading QSAM input file to total & print……..
          EVALUATE HOUSEINV-STATUS
               WHEN '00'
                  MOVE HOUSE-RECORD TO CURRENT-RECORD
                  MOVE SPACES TO PRINT-RECORD
                  MOVE ITEMNO TO PRT-ITEMNO
…………..snip at the end of this program, checking to see if
……………there was a "problem" by checking the error-indicator switch….
          IF ERROR-INDICATOR = 'Y'
                 MOVE +8 TO RETURN-CODE
               ELSE
                 MOVE +0 TO RETURN-CODE
           END-IF.
           GOBACK.
```

This is a portion of the COBOL application that sets up the external data areas "just in case" some problem happens. Notice also that the ERROR-INDICATOR can be used to signal back to the main application that some problem occurred, so that the application can end, but with a non-zero return code.



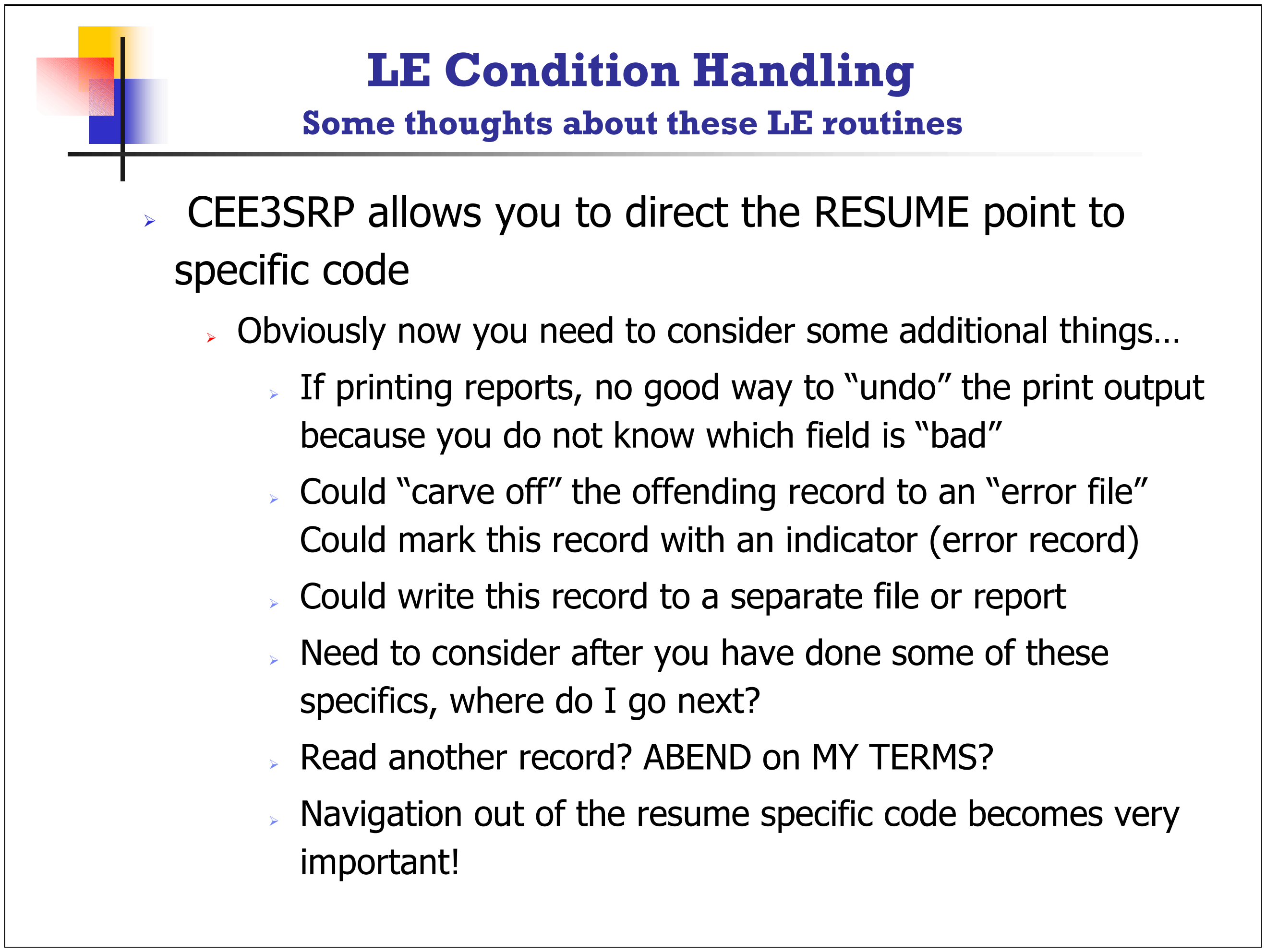
LE Condition Handling

Some thoughts about these LE routines

- Notice the use of the COBOL EXTERNAL elements to "share" data without having to "pass via linkage"
- Any of the variable length elements can be coded as true variable .. These examples used fixed length fields for simplicity and … well…for 3AM programming!

```
01  CONDITION-AREAS             EXTERNAL.
          10  CURRENT-RECORD                      PIC X(80).
          10  ERROR-INDICATOR                     PIC X.
………
01  MSG-STR.
          10  MSG-LENGTH                          PIC S9(4) COMP VALUE +120.
          10  MSG-STRING                          PIC X(120).
Could also be coded:
01  MSG-STR.
          10  MSG-LENGTH                          PIC S9(4) COMP.
          02  MSG-STRING-TEXT.
                 03  MSG-STRING-CHAR              PIC X
                                        OCCURS 0 TO 256 TIMES
                                        DEPENDING ON MSG-LENGTH
                                                  of MSG-STR.
```

Many of the data areas used in Language Environment are variable length elements, and the 3rd example shown here is the "technically correct" way to represent these areas. For simplicity you may want to just code the "string" portion to be long enough for your purposes, and make it a fixed-length data item.

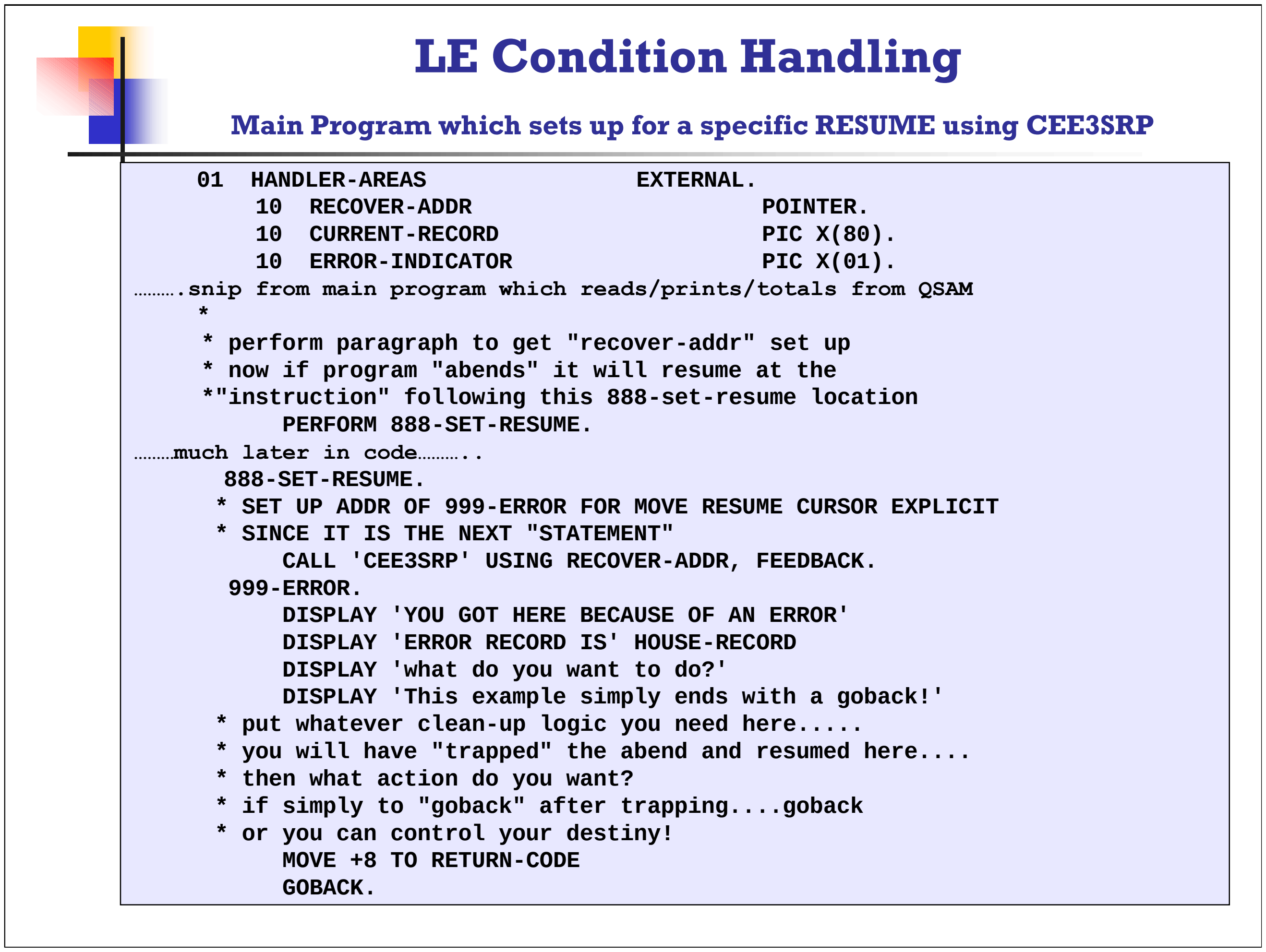


LE Condition Handling

Some thoughts about these LE routines

- CEE3SRP allows you to direct the RESUME point to specific code
  - Obviously now you need to consider some additional things…
    - If printing reports, no good way to "undo" the print output because you do not know which field is "bad"
    - Could "carve off" the offending record to an "error file" Could mark this record with an indicator (error record)
    - Could write this record to a separate file or report
    - Need to consider after you have done some of these specifics, where do I go next?
    - Read another record? ABEND on MY TERMS?
    - Navigation out of the resume specific code becomes very important!

CEE3SRP – Set Resume Point lets you set up the address where you want SPECIFIC RESUMPTION. This takes some thought and PRE-PLANNING, but could prove VERY USEFUL to allow user programming to do some additional processing, or write a record off to another "error file" and then read the next record and continue processing…well…you get the idea.



LE Condition Handling

Main Program which sets up for a specific RESUME using CEE3SRP

```
     01  HANDLER-AREAS                 EXTERNAL.
         10  RECOVER-ADDR                       POINTER.
         10  CURRENT-RECORD                     PIC X(80).
         10  ERROR-INDICATOR                    PIC X(01).
……….snip from main program which reads/prints/totals from QSAM
     *
      * perform paragraph to get "recover-addr" set up
      * now if program "abends" it will resume at the
      *"instruction" following this 888-set-resume location
           PERFORM 888-SET-RESUME.
………much later in code……….
       888-SET-RESUME.
       * SET UP ADDR OF 999-ERROR FOR MOVE RESUME CURSOR EXPLICIT
       * SINCE IT IS THE NEXT "STATEMENT"
           CALL 'CEE3SRP' USING RECOVER-ADDR, FEEDBACK.
        999-ERROR.
           DISPLAY 'YOU GOT HERE BECAUSE OF AN ERROR'
           DISPLAY 'ERROR RECORD IS' HOUSE-RECORD
           DISPLAY 'what do you want to do?'
           DISPLAY 'This example simply ends with a goback!'
       * put whatever clean-up logic you need here.....
       * you will have "trapped" the abend and resumed here....
       * then what action do you want?
       * if simply to "goback" after trapping....goback
       * or you can control your destiny!
           MOVE +8 TO RETURN-CODE
           GOBACK.
```

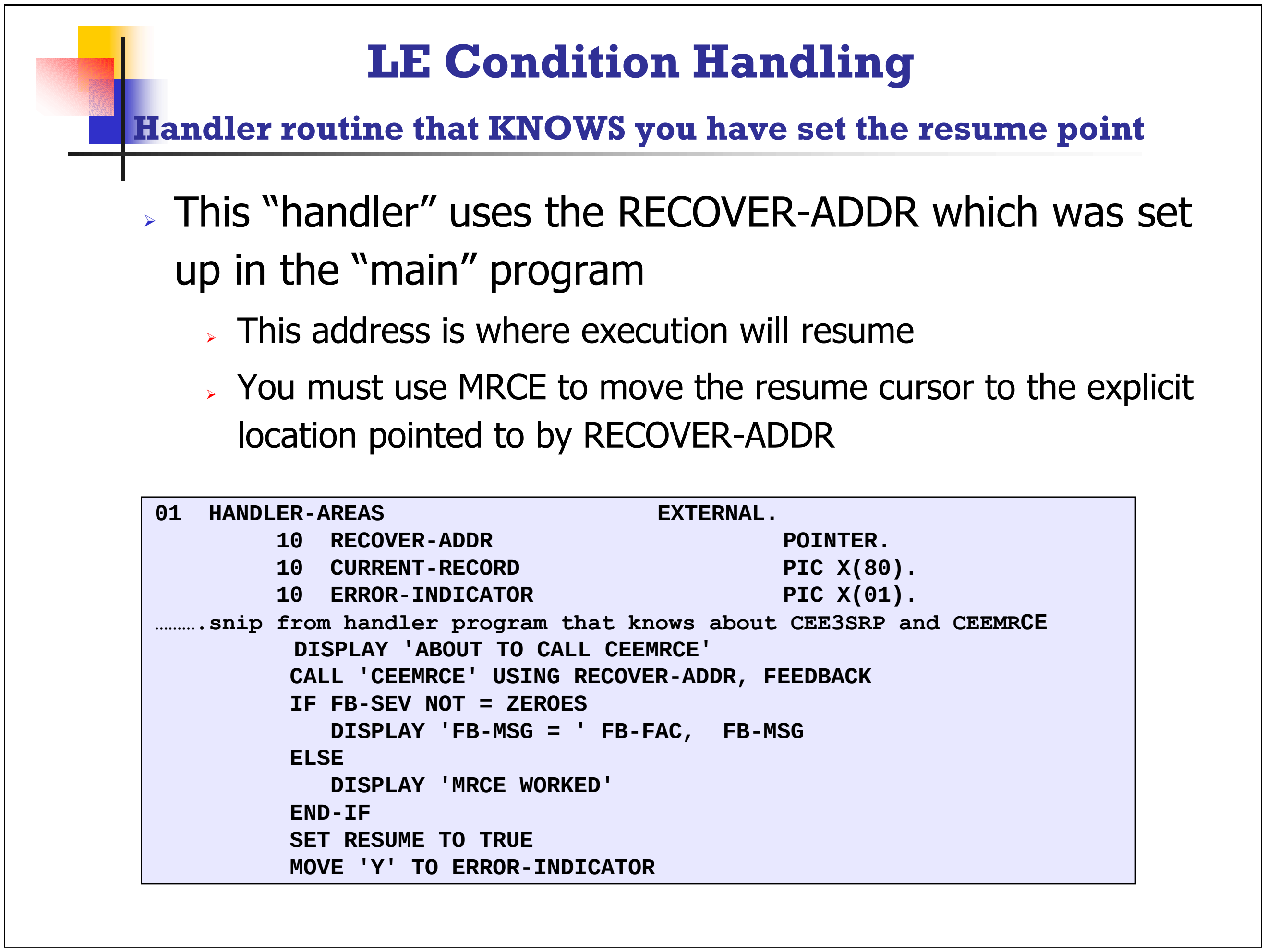
In this example the 888-SET-RESUME paragraph is really a dummy paragraph simply to allow the SRP to be the next set of code. By using a paragraph for the CEESRP, you can PERFORM the 888-SET-RESUME just to get the resume point set up, then in the condition handler when you do the CALL ‘CEEMRCE’ the resume point will be positioned at 999-ERROR.

Note that this example REQUIRES THE COMPILER OPTION NOOPT! If you are OPTIMIZING your code, COBOL will interpret this paragraph as unreachable and unexecutable and discard it, so for the program setting up a specific resume point using this technique, be sure to use

PROCESSS NOOPT

or

CBL NOOPT to force nooptimization!



LE Condition Handling

Handler routine that KNOWS you have set the resume point

- This “handler” uses the RECOVER-ADDR which was set up in the “main” program
  - This address is where execution will resume
  - You must use MRCE to move the resume cursor to the explicit location pointed to by RECOVER-ADDR

```
01  HANDLER-AREAS                    EXTERNAL.
         10  RECOVER-ADDR                       POINTER.
         10  CURRENT-RECORD                     PIC X(80).
         10  ERROR-INDICATOR                    PIC X(01).
……….snip from handler program that knows about CEE3SRP and CEEMRCE
           DISPLAY 'ABOUT TO CALL CEEMRCE'
           CALL 'CEEMRCE' USING RECOVER-ADDR, FEEDBACK
           IF FB-SEV NOT = ZEROES
              DISPLAY 'FB-MSG = ' FB-FAC,  FB-MSG
           ELSE
              DISPLAY 'MRCE WORKED'
           END-IF
           SET RESUME TO TRUE
           MOVE 'Y' TO ERROR-INDICATOR
```

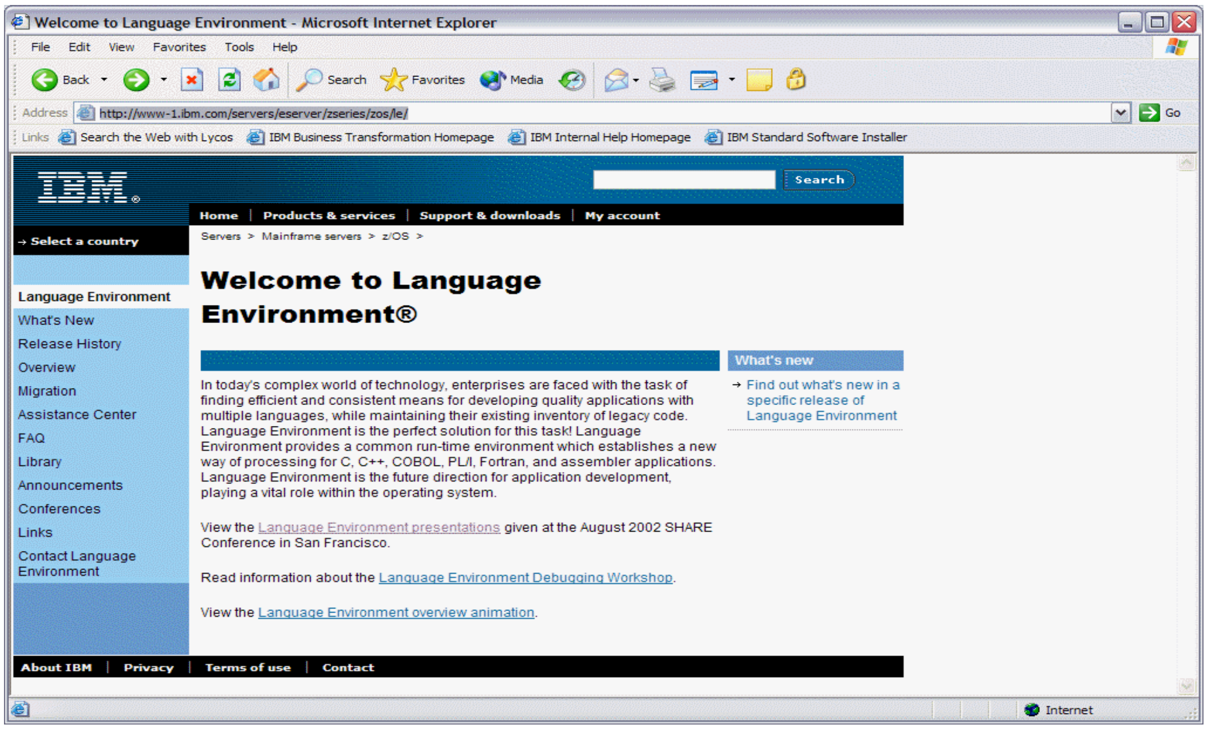
This is a portion of a VERY SIMPLE condition handler that issues the CEEMRCE, move resume cursor EXPLICIT, to use the address in RECOVER-ADDR which was set up in the main application.

LE Condition Handling
A Review, after all this information

- **What?**
  - Condition handlers give you EXTENSIVE capabilities on how you can deal with conditions..you can resume, you can capture and report additional information, you can always abend…lots of routines supplied with LE
- **How?**
  - Your user routines can use LE's capabilities, and the ways you can use LE's services should be clearer after all this
- **Where?**
  - Your routines can get control when some condition occurs AS LONG AS YOU HAVE "REGISTERED" a HANDLER ROUTINE and you can register many handlers, in many ways including via a PARM='/ USRHDLR(progname)'
- **When?**
  - At what point does it make sense to "handle" condition? This is the big question and is VERY SPECIFIC TO your processing and business!
    - For TESTING condition handling is great, for production there are many additional considerations, but now you should have some ideas about all this!

LE Condition Handling

Where you can read more about all this…



Welcome to Language Environment - Microsoft Internet Explorer

Address: http://www-1.ibm.com/servers/eserver/zseries/zos/le/

Links: Search the Web with Lycos | IBM Business Transformation Homepage | IBM Internal Help Homepage | IBM Standard Software Installer

IBM
Search
Home | Products & services | Support & downloads | My account
→ Select a country
Servers > Mainframe servers > z/OS >
Language Environment
What's New
Release History
Overview
Migration
Assistance Center
FAQ
Library
Announcements
Conferences
Links
Contact Language Environment
What's new
About IBM | Privacy | Terms of use | Contact
Internet

Welcome to Language Environment®

In today's complex world of technology, enterprises are faced with the task of finding efficient and consistent means for developing quality applications with multiple languages, while maintaining their existing inventory of legacy code. Language Environment is the perfect solution for this task! Language Environment provides a common run-time environment which establishes a new way of processing for C, C++, COBOL, PL/I, Fortran, and assembler applications. Language Environment is the future direction for application development, playing a vital role within the operating system.

View the Language Environment presentations given at the August 2002 SHARE Conference in San Francisco.

Read information about the Language Environment Debugging Workshop.

View the Language Environment overview animation.

→ Find out what's new in a specific release of Language Environment

The manuals are available by selecting LIBRARY from the main LE web page, and then you can pick either bookmanager of PDF style manuals:

The web site for Language Environment is:

http://www-1.ibm.com/servers/eserver/zseries/zos/le/

From here you can navigate to many places:

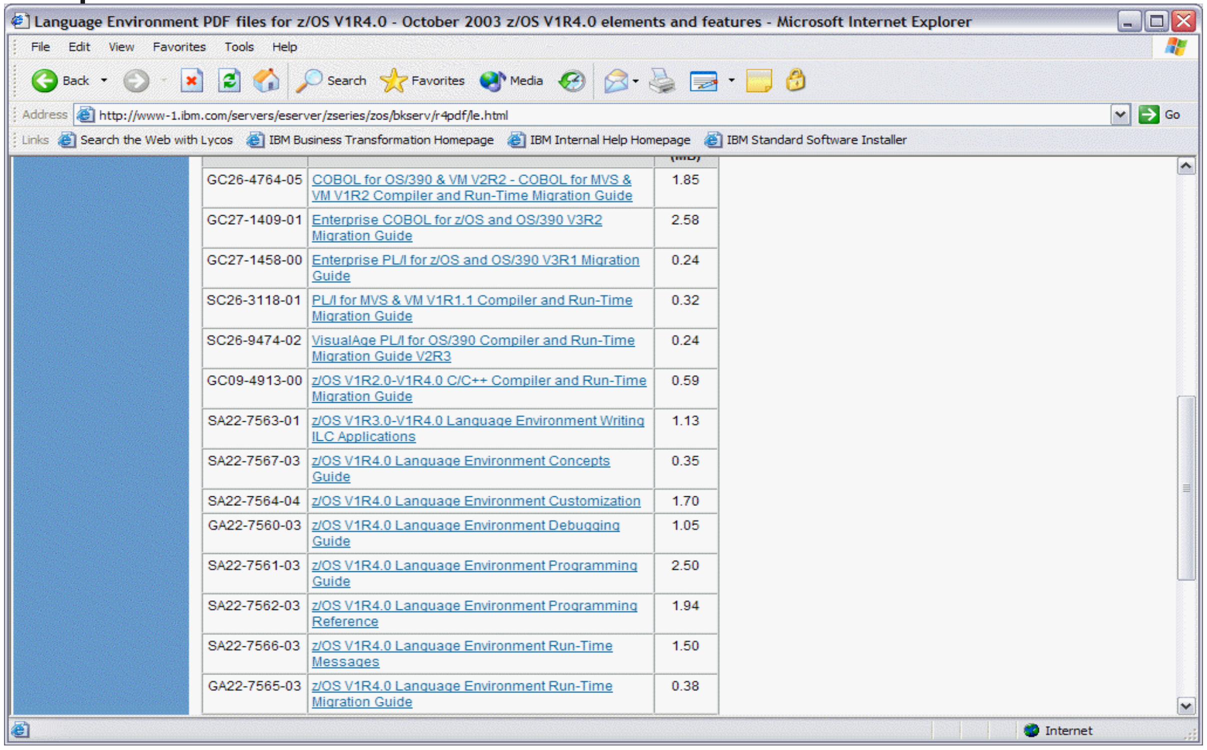
Library – for the manuals

FAQ – frequently asked questions

You can even CONTACT LANGUAGE ENVIRONMENT if you want!

LE Condition Handling

Where you can read more about all this…



File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Address
Go
Links
Search the Web with Lycos
IBM Business Transformation Homepage
IBM Internal Help Homepage
IBM Standard Software Installer
Internet

Language Environment PDF files for z/OS V1R4.0 - October 2003 z/OS V1R4.0 elements and features - Microsoft Internet Explorer

http://www-1.ibm.com/servers/eserver/zseries/zos/bkserv/r4pdf/le.html

GC26-4764-05	COBOL for OS/390 & VM V2R2 - COBOL for MVS & VM V1R2 Compiler and Run-Time Migration Guide	1.85
GC27-1409-01	Enterprise COBOL for z/OS and OS/390 V3R2 Migration Guide	2.58
GC27-1458-00	Enterprise PL/I for z/OS and OS/390 V3R1 Migration Guide	0.24
SC26-3118-01	PL/I for MVS & VM V1R1.1 Compiler and Run-Time Migration Guide	0.32
SC26-9474-02	VisualAge PL/I for OS/390 Compiler and Run-Time Migration Guide V2R3	0.24
GC09-4913-00	z/OS V1R2.0-V1R4.0 C/C++ Compiler and Run-Time Migration Guide	0.59
SA22-7563-01	z/OS V1R3.0-V1R4.0 Language Environment Writing ILC Applications	1.13
SA22-7567-03	z/OS V1R4.0 Language Environment Concepts Guide	0.35
SA22-7564-04	z/OS V1R4.0 Language Environment Customization	1.70
GA22-7560-03	z/OS V1R4.0 Language Environment Debugging Guide	1.05
SA22-7561-03	z/OS V1R4.0 Language Environment Programming Guide	2.50
SA22-7562-03	z/OS V1R4.0 Language Environment Programming Reference	1.94
SA22-7566-03	z/OS V1R4.0 Language Environment Run-Time Messages	1.50
GA22-7565-03	z/OS V1R4.0 Language Environment Run-Time Migration Guide	0.38

The manuals are available by selecting LIBRARY from the main LE web page, and then you can pick either bookmanager of PDF style manuals:

You can download whichever version you want, including the Vendor Interfaces manual (the last line on this screen) which contains a great deal of “inside” information! You can pick the level of LE and get all the manuals for your specific release!