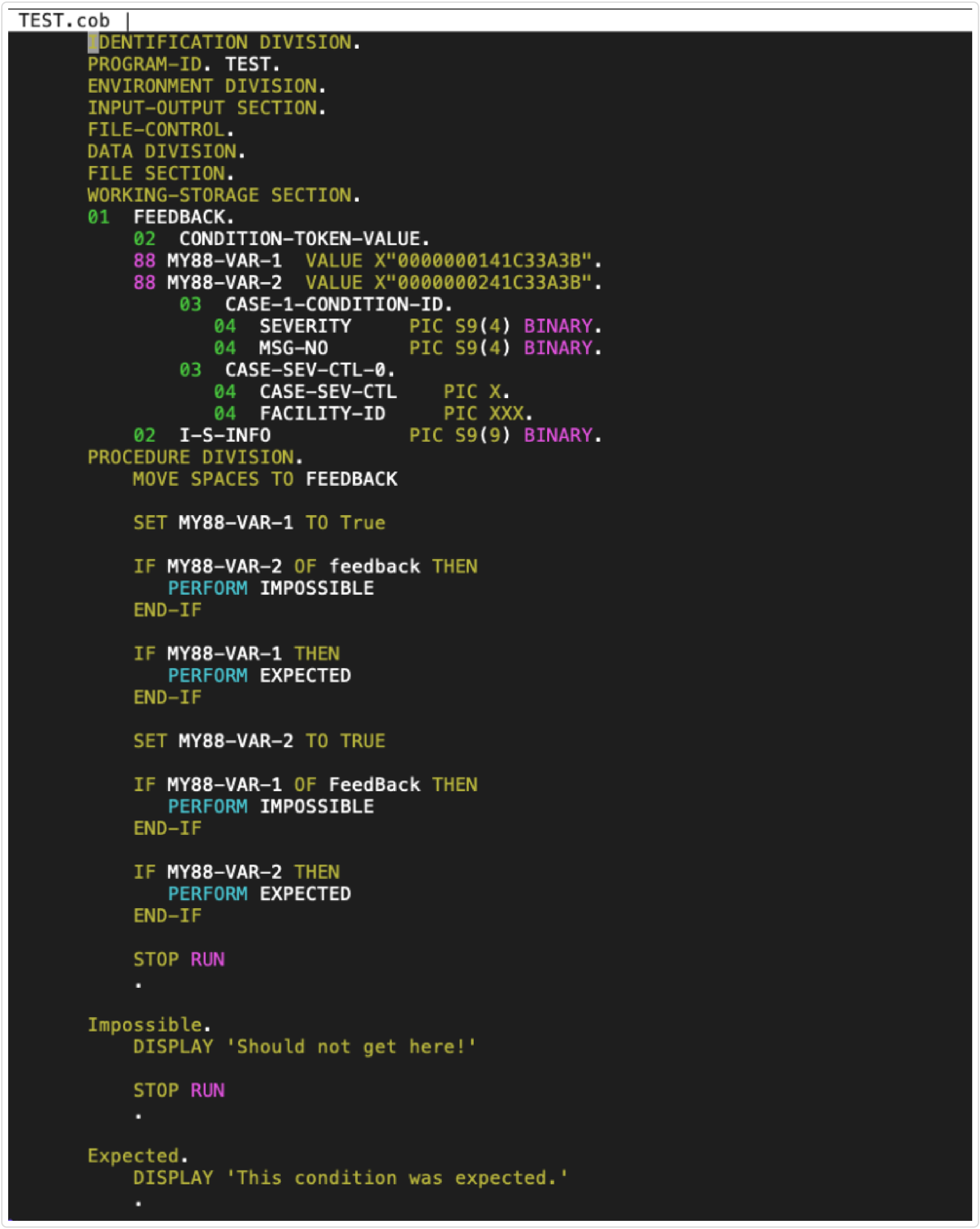
(https://peeterjoot.com) Using the debugger to understand COBOL level 88 semantics.

August 10, 2020 COBOL (https://peeterjoot.com/category/mainframe/cobol/)
COBOL (https://peeterjoot.com/tag/cobol/), LEVEL 88 (https://peeterjoot.com/tag/level-88/), lldb (https://peeterjoot.com/tag/lldb/)

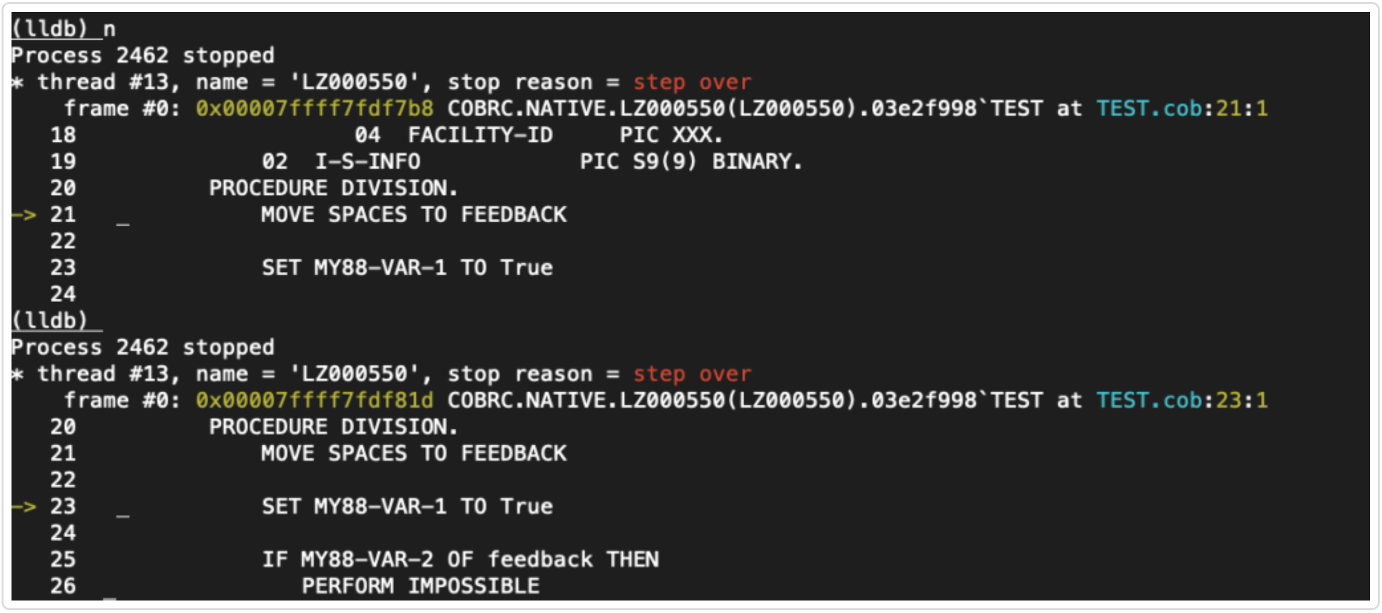
COBOL has an enumeration mechanism called a LEVEL-88 variable. I found a few aspects of this counter-intuitive, as I've mentioned in a few previous posts. With the help of the debugger, I now think I finally understand what's going on. Here's an example program that uses a couple of LEVEL-88 variables:



```
TEST.cob |
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01  FEEDBACK.
           02  CONDITION-TOKEN-VALUE.
           88 MY88-VAR-1  VALUE X"0000000141C33A3B".
           88 MY88-VAR-2  VALUE X"0000000241C33A3B".
               03  CASE-1-CONDITION-ID.
                   04  SEVERITY     PIC S9(4) BINARY.
                   04  MSG-NO       PIC S9(4) BINARY.
               03  CASE-SEV-CTL-0.
                   04  CASE-SEV-CTL    PIC X.
                   04  FACILITY-ID     PIC XXX.
           02  I-S-INFO             PIC S9(9) BINARY.
       PROCEDURE DIVISION.
           MOVE SPACES TO FEEDBACK

           SET MY88-VAR-1 TO True

           IF MY88-VAR-2 OF feedback THEN
              PERFORM IMPOSSIBLE
           END-IF

           IF MY88-VAR-1 THEN
              PERFORM EXPECTED
           END-IF

           SET MY88-VAR-2 TO TRUE

           IF MY88-VAR-1 OF FeedBack THEN
              PERFORM IMPOSSIBLE
           END-IF

           IF MY88-VAR-2 THEN
              PERFORM EXPECTED
           END-IF

           STOP RUN
           .

       Impossible.
           DISPLAY 'Should not get here!'

           STOP RUN
           .

       Expected.
           DISPLAY 'This condition was expected.'
           .
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-11.28.50-AM.png)

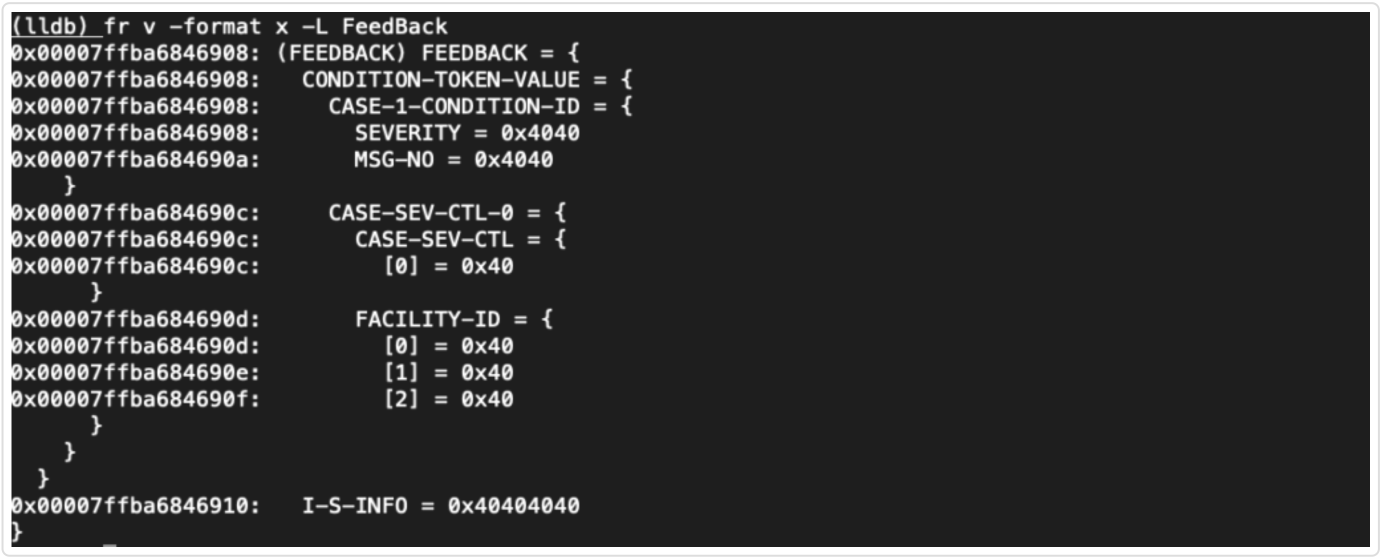
We can use a debugger to discover the meaning of the COBOL code. Let’s start by single stepping past the first MOVE statement to just before the SET MY88-VAR-1:



```
(lldb) n
Process 2462 stopped
* thread #13, name = 'LZ000550', stop reason = step over
    frame #0: 0x00007ffff7fdf7b8 COBRC.NATIVE.LZ000550(LZ000550).03e2f998`TEST at TEST.cob:21:1
   18                       04  FACILITY-ID      PIC XXX.
   19                   02  I-S-INFO            PIC S9(9) BINARY.
   20               PROCEDURE DIVISION.
-> 21   _               MOVE SPACES TO FEEDBACK
   22
   23                   SET MY88-VAR-1 TO True
   24
(lldb)
Process 2462 stopped
* thread #13, name = 'LZ000550', stop reason = step over
    frame #0: 0x00007ffff7fdf81d COBRC.NATIVE.LZ000550(LZ000550).03e2f998`TEST at TEST.cob:23:1
   20               PROCEDURE DIVISION.
   21                   MOVE SPACES TO FEEDBACK
   22
-> 23   _               SET MY88-VAR-1 TO True
   24
   25                   IF MY88-VAR-2 OF feedback THEN
   26  _                   PERFORM IMPOSSIBLE
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-11.45.38-AM.png)

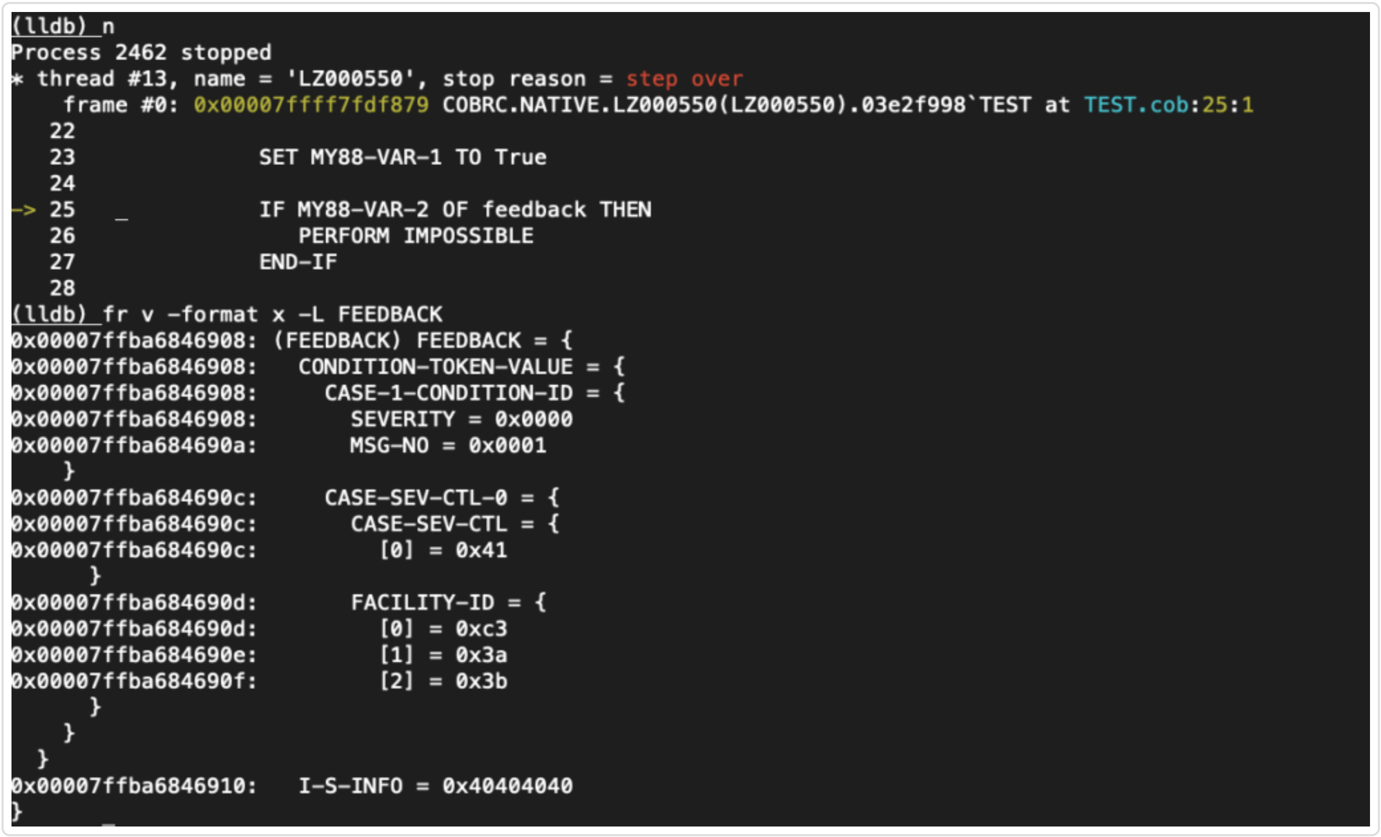
Here, I'm running the program LZ000550 from a PDS named COBRC.NATIVE.LZ000550. We expect EBCDIC spaces ('\x40') in the FEEDBACK structure at this point and that's what we see:



```
(lldb) fr v -format x -L FeedBack
0x00007ffba6846908: (FEEDBACK) FEEDBACK = {
0x00007ffba6846908:   CONDITION-TOKEN-VALUE = {
0x00007ffba6846908:     CASE-1-CONDITION-ID = {
0x00007ffba6846908:       SEVERITY = 0x4040
0x00007ffba684690a:       MSG-NO = 0x4040
    }
0x00007ffba684690c:     CASE-SEV-CTL-0 = {
0x00007ffba684690c:       CASE-SEV-CTL = {
0x00007ffba684690c:         [0] = 0x40
      }
0x00007ffba684690d:       FACILITY-ID = {
0x00007ffba684690d:         [0] = 0x40
0x00007ffba684690e:         [1] = 0x40
0x00007ffba684690f:         [2] = 0x40
      }
    }
  }
0x00007ffba6846910:   I-S-INFO = 0x40404040
}
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-11.48.59-AM.png)

Line stepping past the SET statement, our structure memory layout now looks like:



```
(lldb) n
Process 2462 stopped
* thread #13, name = 'LZ000550', stop reason = step over
    frame #0: 0x00007ffff7fdf879 COBRC.NATIVE.LZ000550(LZ000550).03e2f998`TEST at TEST.cob:25:1
   22
   23               SET MY88-VAR-1 TO True
   24
-> 25   _           IF MY88-VAR-2 OF feedback THEN
   26                  PERFORM IMPOSSIBLE
   27               END-IF
   28
(lldb) fr v -format x -L FEEDBACK
0x00007ffba6846908: (FEEDBACK) FEEDBACK = {
0x00007ffba6846908:   CONDITION-TOKEN-VALUE = {
0x00007ffba6846908:     CASE-1-CONDITION-ID = {
0x00007ffba6846908:       SEVERITY = 0x0000
0x00007ffba684690a:       MSG-NO = 0x0001
    }
0x00007ffba684690c:     CASE-SEV-CTL-0 = {
0x00007ffba684690c:       CASE-SEV-CTL = {
0x00007ffba684690c:         [0] = 0x41
      }
0x00007ffba684690d:       FACILITY-ID = {
0x00007ffba684690d:         [0] = 0xc3
0x00007ffba684690e:         [1] = 0x3a
0x00007ffba684690f:         [2] = 0x3b
      }
    }
  }
0x00007ffba6846910:   I-S-INFO = 0x40404040
}
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-11.51.30-AM.png)

By setting the LEVEL-88 variable to TRUE all the memory starting at the address of feedback is now overwritten by the numeric value 0x0000000141C33A3BL. If we continue the program, the SYSOUT ends up looking like:

```
This condition was expected.
This condition was expected.
```

The first ‘IF MY88-VAR-1 THEN’ fires, and after the subsequent ‘SET MY88-VAR-2 TO TRUE’, the second ‘IF MY88-VAR-2 THEN’ fires. The SET overwrites the structure memory at the point that the 88 was declared, and an IF check of that same variable name checks if the memory in that location has the value in the 88 variable. It does not matter what the specific layout of the structure is at that point. We see that an IF check of the level-88 variable just tests whether or not the value at that address has the pattern specified in the variable. In this case, we have only on level-88 variable with the given name in the program, so the ‘IF MY88-VAR-2 OF feedback’ that was used was redundant, and could have been coded as just ‘IF MY88-VAR-2’, or could have been coded as ‘IF MY88-VAR-2 OF CONDITION-TOKEN-VALUE of Feedback’

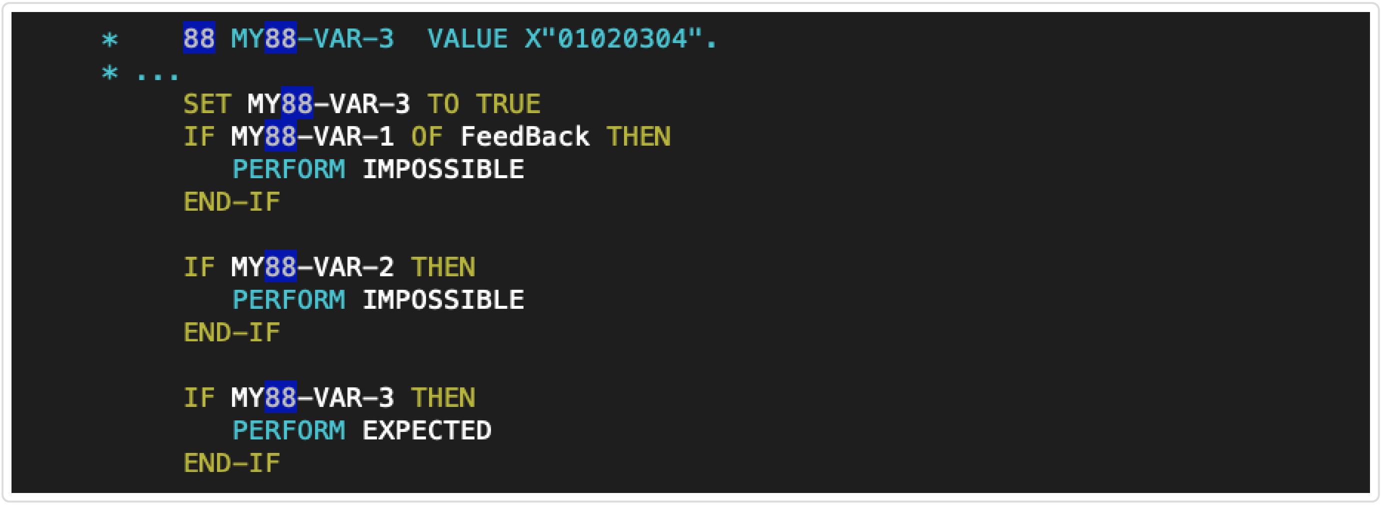
We can infer that the COBOL code’s WORKING-STORAGE has the following equivalent C++ layout:

```
struct CONDITION_TOKEN_VALUE
{
   short SEVERITY;
   short MSG_NO;
   char CASE_SEV_CTL;
   char FACILITY_ID[3];
};

enum my88_vars
{
   MY88_VAR_1 = 0x0000000141C33A3BL,
   MY88_VAR_2 = 0x0000000241C33A3BL
};

struct feedback
{
   union {
      CONDITION_TOKEN_VALUE c;
      my88_vars e;
   } u;
   int I_S_INFO;
};
```

and that the control flow of the program can be modeled as the following:

```
//...
feedback f;

int main()
{
   memset( &f, 0x40, sizeof(f) );
   f.u.e = MY88_VAR_1;
   if ( f.u.e == MY88_VAR_2 )
   {
      impossible();
   }
   if ( f.u.e == MY88_VAR_1 )
   {
      expected();
   }

   f.u.e = MY88_VAR_2;
   if ( f.u.e == MY88_VAR_1 )
   {
      impossible();
   }
   if ( f.u.e == MY88_VAR_2 )
```

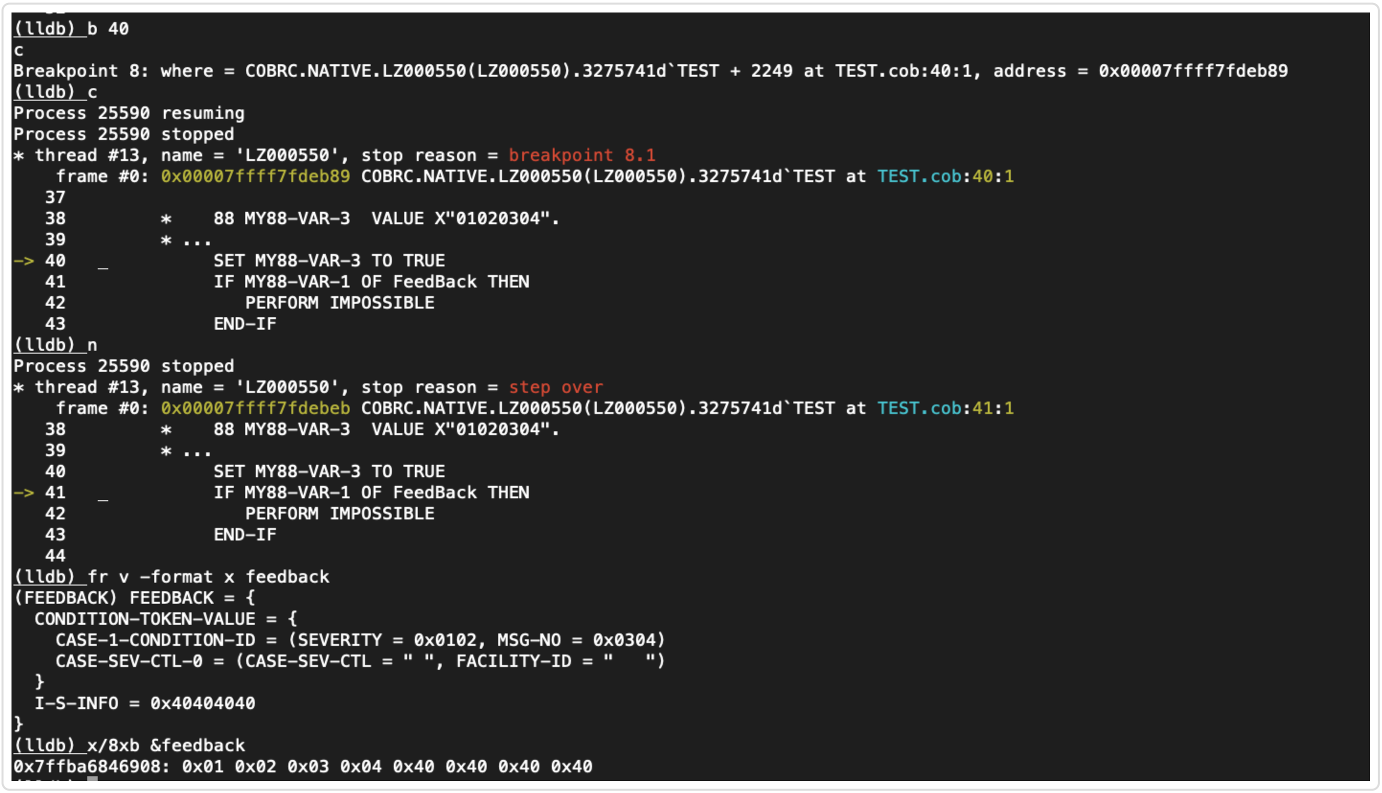
Things also get even more confusing if the LEVEL-88 variable specifies less bytes than the structure that it is embedded in. In that case, SET of the variable pads out the structure with spaces and a check of the variable also looks for those additional trailing spaces:



```
      *    88 MY88-VAR-3  VALUE X"01020304".
      * ...
           SET MY88-VAR-3 TO TRUE
           IF MY88-VAR-1 OF FeedBack THEN
              PERFORM IMPOSSIBLE
           END-IF

           IF MY88-VAR-2 THEN
              PERFORM IMPOSSIBLE
           END-IF

           IF MY88-VAR-3 THEN
              PERFORM EXPECTED
           END-IF
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-12.34.27-PM.png)

The CONDITION-TOKEN-VALUE object uses a total of 8 bytes. We can see the spaces in the display of the FEEDBACK structure if we look in the debugger:



```
(lldb) b 40
c
Breakpoint 8: where = COBRC.NATIVE.LZ000550(LZ000550).3275741d`TEST + 2249 at TEST.cob:40:1, address = 0x00007ffff7fdeb89
(lldb) c
Process 25590 resuming
Process 25590 stopped
* thread #13, name = 'LZ000550', stop reason = breakpoint 8.1
    frame #0: 0x00007ffff7fdeb89 COBRC.NATIVE.LZ000550(LZ000550).3275741d`TEST at TEST.cob:40:1
   37
   38         *    88 MY88-VAR-3  VALUE X"01020304".
   39         * ...
-> 40   _          SET MY88-VAR-3 TO TRUE
   41              IF MY88-VAR-1 OF FeedBack THEN
   42                 PERFORM IMPOSSIBLE
   43              END-IF
(lldb) n
Process 25590 stopped
* thread #13, name = 'LZ000550', stop reason = step over
    frame #0: 0x00007ffff7fdebeb COBRC.NATIVE.LZ000550(LZ000550).3275741d`TEST at TEST.cob:41:1
   38         *    88 MY88-VAR-3  VALUE X"01020304".
   39         * ...
   40              SET MY88-VAR-3 TO TRUE
-> 41   _          IF MY88-VAR-1 OF FeedBack THEN
   42                 PERFORM IMPOSSIBLE
   43              END-IF
   44
(lldb) fr v -format x feedback
(FEEDBACK) FEEDBACK = {
  CONDITION-TOKEN-VALUE = {
    CASE-1-CONDITION-ID = (SEVERITY = 0x0102, MSG-NO = 0x0304)
    CASE-SEV-CTL-0 = (CASE-SEV-CTL = " ", FACILITY-ID = "   ")
  }
  I-S-INFO = 0x40404040
}
(lldb) x/8xb &feedback
0x7ffba6846908: 0x01 0x02 0x03 0x04 0x40 0x40 0x40 0x40
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-12.36.07-PM.png)

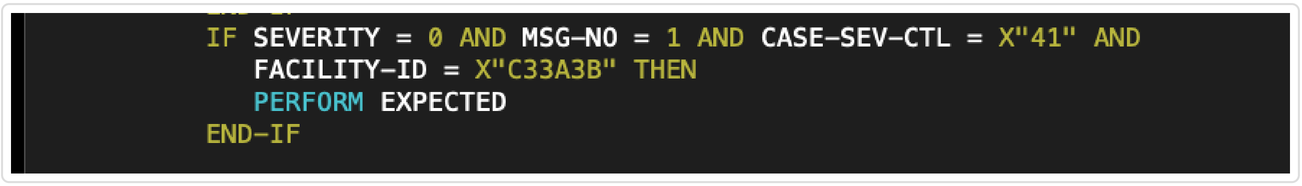
See the four trailing 0x40 spaces here.

Incidentally, it can be hard to tell what the total storage requirements of a COBOL structure is by just looking at the code, because the mappings between digits and storage depends on the usage clauses. If the structure also uses REDEFINES clauses (embedded unions), as was the case in the program that I was originally looking at, the debug output is also really nice to understand how big the various fields are, and where they are situated.

Here are a few of the lessons learned:

- You might see a check like ‘IF MY88-VAR-1 THEN’, but nothing in the program explicitly sets MY88-VAR-1. It is effectively a global variable value that is set as a side effect of some other call (in the real program I was looking at, what modified this “variable” was actually a call to CEEFMDA, a LE system service.) We have pass by reference in the function calls so it can be a reverse engineering task to read any program and figure out how any given field may have been modified, and that doesn’t get any easier by introducing LEVEL-88 variables into the mix.

- This effective enumeration mechanism is not typed in any sense. Correct use of a LEVEL-88 relies on the variables that follow it to have the appropriate types. In this case, the ‘IF MY88-VAR-1 THEN’ is essentially shorthand for:



```
           IF SEVERITY = 0 AND MSG-NO = 1 AND CASE-SEV-CTL = X"41" AND
              FACILITY-ID = X"C33A3B" THEN
              PERFORM EXPECTED
           END-IF
```

(https://peeterjoot.com/wp-content/uploads/2020/08/Screen-Shot-2020-08-10-at-12.22.37-PM.png)

- There is a disconnect between the variables modified or checked by a given LEVEL-88 variable reference that must be inferred by the context.
- An IF check of a LEVEL-88 variable may include an implicit check of the trailing part of the structure with EBCDIC spaces, if the fields that follow the 88 variable take more space than the value of the variable. Similarly, seting such a variable to TRUE may effectively memset a trailing subset of the structure to EBCDIC spaces.
- Exactly what is modified by a given 88 variable depends on the context. For example, if the level 88 variables were found in a copybook, and if I had a second structure that had the same layout as FEEDBACK, with both structures including that copybook, then I’d have two instances of this “enumeration”, and would need a set of “OF foo OF BAR” type clauses to disambiguate things. Level 88 variables aren’t like a set of C defines. Their meaning is context dependent, even if they masquerade as constants.

Back to top