ZoneCheck

(Friend, Foe, or Existential Enigma)

William M. Klein

wmklein@ix.netcom.com

Contents

Why and when to use ZoneCheck

The rules of COBOL have always stated that when a numeric field contains invalid data and is used as a sending field, the "results are unpredictable." This is true for the ANSI and ISO COBOL Standards and for every version of IBM COBOL. (See "invalid data" in the LRM.) Because IBM has always provide a high degree of upward compatibility between COBOL products, many users have not paid attention to this unpredictability. They often have relied on the object code ABENDing with certain invalid data while continuing with processing under certain other language constructs. This dependency on specific processing first became obvious when users started moving from OS/VS COBOL to VS COBOL II. Users expected the same run-time behavior for numeric fields with invalid data with the newer (redesigned) VS COBOL II compiler. In order to reduce this migration inhibitor, IBM introduced the NUMPROC(MIG) compiler option which provided similar results with VS COBOL II has had happened with OS/VS COBOL. However, this was expected to be a migration tool and it was thought that most programmers would change their code to better process invalid data so that they could gradually move to NUMPROC(PFD) which has better performance behavior.

When IBM next redesigned their compiler to better take advantage of z/OS features with Enterprise COBOL V5, the NUMPROC(MIG) facility was removed and it was discovered that many applications still had dependencies on specific processing of numeric fields with invalid data contents. Rather than re-introducing the NUMPROC(MIG) feature (and encouraging poor run-time performance), IBM introduced the ZoneCheck compiler option as a tool for detecting and correcting the handling of zone decimal numeric fields with invalid content.

Therefore, before moving a COBOL production application to Enterprise COBOL V6 (or V5), it is important to use the ZoneCheck compiler option (along with NumProc(PFD) so the COBOL code does not rely on any specific behavior for the processing of numeric fields with invalid data.

Once, you have completed your full testing of V6 programs with ZoneCheck,NumProc(PFD), you will be able to use NoZoneCheck and NUMPROC(PFD) to get the best performance of your production code while still obtaining the expected run-time results. It is possible to test with ZoneCheck,NumProc(NoPFD) and then run your production code with NumProc(NoPFD). this will not get the same performance boost as using NumProc(PFD), but may make your migration easier.

NOTE: The information below assumes that you are doing a migration from Enterprise COBOL V4.2 to Enterprise COBOL V6 (or V5). If you are not already using V4.2, it is recommended that you migrate to that level before trying to do a migration to V6. (Some sites have found it beneficial to skip this initial migration to V4.2. This means that you have one less migration and testing step. However, it also means that you do not have a V4.2 version of your application to fall back on, nor will you be able to compare V4.2 and V6 output results. Each site will need to weigh the PROs and CONs of doing an initial migration to V4.2)

There is also a PTF (UI32232) that allows you to use ZoneCheck with V4.2. Therefore, you can do a code upgrade to handle invalid data without moving to the newer compiler. However, this requires significant testing resources without gaining the performance enhancements of the newer compiler. Much of the following strategies would be useful if you are doing such a code upgrade without also doing a compiler upgrade.

The ultimate goal of your use of ZoneCheck is to create a V6 object code compiled with ZoneCheck(Msg) and NumProc(PFD) that can be run with a full set of production input that produces the same output as the source code compiled with V4.2 and produces no ZoneCheck messages. Once you reach this stage, you can recompile with NoZoneCheck , NumProc(PFD), and Opt(2) using the V6 compiler and running the resultant object code safely in production.

Before you begin your migration

It is suggested that you try to create source code that can be compiled successfully with both Enterprise COBOL V4.2 and V6. If you do this, it will make your testing easier and will also provide an easier fall back should the V6 version have difficulties in production. Therefore, if the reason that you are doing a migration on this specific program is to add or change functionality, it is recommended that you do not use any of the new and enhanced features of the V6 compiler until you have verified that the V6 and V4.2 compilers produce identical output when receiving the same input.

If you compile your V4.2 source code with the FLAGMIG4 and FLAG(I,I) compiler options, you should be able to validate that the source code will compile cleanly with V6.

You should also compile your V6 code with the FLAG(I,I),DIAGTRUNC and RULES(NoEvenPack) compiler options. These options will tell you if you have any constructs in your source code that may lead to truncated or misinformation. It is recommended that you fix these situations before attempting to resolve ZoneCheck issues. the use of the FLAG(I,I) compiler option will also show you if there are any other informational messages that need resolving to insure proper program logic.

It should be noted that both the FLAGMIG4 and DIAGTRUNC compiler options may be omitted by some programmers as they either provide too many or too few messages to be truly useful. As you use them with more programs, you should be able to determine what you think is most useful for your applications.

Once you have the source code without any of these problems you can move to testing with ZoneCheck. It is possible to fix multiple problems at a single time, however, this approach often leads to requiring more resources in the long run.

Testing Strategies

The most important issue with using the ZoneCheck compiler option is that you do sufficient testing to insure that your revised program runs successfully and creates the proper output in production. To do this it is suggested that you do a staged testing.

- First, do your usual unit testing for this individual program
- Next, do your usual unit testing for the entire application of which this program is a part
- Next, Do your usual quality testing with as wide a variety of test input as is available
- Finally, either run the revised V6 program in parallel with your V4.2 version of the program in production or, if available, run both the V4.2 and V6 versions of your program in a test environment with multiple sets of production input.

In each stage, you want to make certain that your V6 program compiled with ZoneCheck(Msg),NumProcPFD) does not issue any messages and that your V4.2 and V6 programs produce identical output.

Most sites require and it is highly recommended that after you have resolved all these issues, that you compile your program with the compiler options to be used in production and then run a final test with that object code. This will insure that there are no object code changes introduced by production compiler options that might impact your final results.

If you are revising a batch or IMS program, you may find it useful to consider the LE MSGFILE run-time option to direct your ZoneCheck output to a specific file for sorting or looking at unprintable characters in hex. If you are revising a CICS program, you will need to work with the CESE TQ to get your ZoneCheck messages.

The messages can be particularly useful if you have access to an interactive debugger such as Debug Tool. The ZoneCheck messages tell you the specific line in your source code or specific fields that you will want to look at while debugging your program.

Questions to ask about your ZoneCheck messages

There are a number of questions that you will want to ask and answer using the ZoneCheck(Msg) output. Depending upon the variety and number of input items to your test, there may be a few or many ZoneCheck messages for a specific test of your program. If there are only a few messages you should be able to use your output either via an online output viewer or from printed output. However, if there are many messages as may happen with a complex program or with a large amount of test input data, then it is recommended that you create and use a message dataset as mentioned above.

One thing to remember while looking at your ZoneCheck messages is that, unlike the NumProc(NOPFD) compiler option, the ZoneCheck compiler option does no actual content correction. Therefore, it is relatively common for the same field to appear in multiple messages until the content of that field is modified by your program logic.

Once you have your ZoneCheck messages in a usable form, you will want to consider and answer the following questions:

- For a specific line in your program, is there a message for one field's content? for all field's content? for a few or for many field's content?
- Are there messages for the same field on many lines? only on a single line? on many but not all lines?
- Are the contents invalid because the entire field is composed of spaces? low-values (binary zeroes)? high-values? EBCDIC character strings?
- Are the contents numeric except that the sign nibble doesn't match the PICTURE clause?
- Is there a single byte (or a few bytes) within the field that contain spaces, binary zeroes, or unprintable characters while the other bytes contain valid numeric digits?
- Is the field either explicitly or implicitly redefined? Do the contents match the description in the redefinition but not for the field as used in this statement?
- Looking at your entire source code program logic, can you find where and how the fields with invalid data are populated? via a read of a dataset? from a database? from a program that calls this program? from a program that this program calls? from an arithmetic or data manipulation statement within this program?

Once you have the answers to the questions above, it should be relatively clear what corrective action is needed. If the invalid data comes from another program or from an external place such as a database or dataset, then, if possible, you should take the corrective action in the program that creates the original content. If this is not possible, you can still correct it in this program in order to move the program to V6.

Fixing Specific Invalid Data situations

Possibly the easiest way to correct invalid data that is identified by ZoneCheck is to add an IF NUMERIC test immediately before any statement that gets a ZoneCheck message. If the test fails, you can move an arbitrary value, such as ZEROES to the field or do a logic exit from the loop that has the invalid content. This approach is NOT recommended. Not only might this skip valid input, but a zero input value may produce skewed and incorrect final output. Adding such a NUMERIC test at each statement where a message appears may also cause a performance degradation.

A slightly more efficient approach would be to add such a code snippet where the field is first populated. Unfortunately, this too might well lead to incorrect final output.

The following are some specific suggestions on how to correct some types of invalid data contents in a way that is most likely to result in compatible output to that which you receive with V4.2. For each suggestion, it is recommended that you follow your program logic and find the place where the field is first populated. Place the numeric test with the suggested corrective action immediately after the field receives the invalid content. It is possible that this will result in changed run-time behavior because of changes to statements where the field is not used as a numeric sending item. Therefore, it is critical that you closely inspect all final output to insure that it matches what is expected.

The following list is not exhaustive but represents many of the cases that ZoneCheck will detect.

Special Values – Space

If your interactive testing or your batch ZoneCheck output indicates that a numeric field contains all spaces, it is most likely that the previous version of your program treated it as containing zeroes. Therefore, you should add a numeric class test and if that fails, you should check the field for spaces. As the rules of COBOL do not allow a direct test of a numeric field for an alphabetic value such as spaces, the following code example shows how you can use reference modification to do this

```
05  NumField   Pic S9(3)V99.
   . . .
If NumField Numeric
    Continue
Else
```

```
    If NumField (1:) = Spaces
      Move Zeroes to NumField
    Else
        Perform Process-Bad-Data
    End-if
End-if
```

One possible trap that you will need to avoid is adding a BLANK WHEN ZERO clause or changing all 9's to Z's in the PICTURE string. Although that would change the field to consider spaces as a valid value, it would also change the category of the field from numeric to numeric-edited. This is highly likely to result in unintended side effects.

<u>Special Values – Low-Values or High-Values</u>

You can use the same technique as demonstrated above to determine if a field contains all low-values or high-values and then take corrective action (such as moving zeroes to the field). However, if you find a situation where a field contains one of these values but it only occurs once in your test run, then you should examine the program logic to determine if the field is actually being used as a switch. In older programs, it was a relatively common technique to use low-values or high-values in an arbitrary field to indicate initial or terminating processing. If this is what is happening in this case, rather than correcting the value in the numeric field, you should add a new alphanumeric field and use it as a switch for such processing.

Another case when a field's content may be low-values is when a field has not been initialized or when program logic gets to this point in the program without correctly populating the field's content. If this is the case, you should check your program logic flow and determine if the field's content should be set to zero or if the program logic should be modified to not reach this point until a dataset or database record is retrieved.

<u>Bad Sign Nibbles – Unsigned vs Signed</u>

Especially if you are migrating from NumProc(noPFD) use with an earlier compiler to the more efficient NumProc(PFD) compiler option with V6, one of the most common situations that ZoneCheck will detect is a mismatch between the data content and the sign (or lack of sign) in the Picture clause. In other words, defining the data item with an "S" in the Picture clause but having a hex "F" in the data content as the sign nibble or, the reverse, having no "S" in the Picture clause but having either a "C" or "D" as the the sign nibble in the data content.

If your ZoneCheck messages or interactive debugging uncovers this situation, you should examine your program logic to determine if you can safely change the Picture clause; adding or removing an "S. In some cases, this will resolve the problem . However, in many cases you will discover that there are some places in your program that expect a signed value and other places that expect the value to be unsigned – for the same field. In that case, you should have the Picture clause match the most common usage. You can use code such as the following to create a second (redefined) field whose sign matches the Picture clause.

In the following example, assume that NumField was already in your program (as an unsigned field) while NumField-Redef has been added to support a signed field.

```
05 NumField   Pic 9(3)V99.
05 NumField-Redef  S9(3)V99 redefines NumField.
 . . .
77   Temp-NumField   S9(3)V99.
 . . .
If NumField Numeric
    Continue
Else
    If NumField-Redef Numeric
        Move NumField-Redef to Temp-NumField
        Move Temp-NumField to NumField
    Else
        Perform Process-Bad-Data
    End-If
End-IF
```

Notice that this code uses a separate temporary field rather than moving data from one definition directly to the redefinition. Although this may work with some versions of the compiler, the rules of COBOL state that moving data to an overlapping part of storage is undefined. Therefore, using the temporary field insures continued support for this code in the future.

A less common situation that ZoneCheck may detect is if you have a hex "A", "B", or "E" as the sign nibble. If ZoneCheck issues a warning for such a situation, you should contact the systems programmer who installed V6. The chances are that they used the NumCls(ALT) compiler option with V4.2 but used NumCls(Prim) with V6. This is an option that can only be set at installation time and not as a compile-time option.

NOTE: if you do your development and initial testing on a PC or under Unix, you should be aware that these environments, by default, use the same sign nibble for unsigned and positive display numeric fields. Thus there may not be a problem with having or not having an S in the Picture clause for unsigned values. Most debuggers for mainframe applications have an option to override this default. However, for best debugging results, you should verify that such an option is in effect.

Single byte problems

You may notice that some fields include primarily good numeric data. However, they have one byte (or a few bytes) with invalid data – especially spaces or low-values. The most common situation that will cause this problem is that, when the field is initially populated, it has valid numeric data, but somewhere during the program logic manipulation of an explicit or implicit redefinition of the field corrupts some of the data in the field. If this is the case, you should modify your program logic so that this field continues to have valid numeric data. You may need to create a temporary copy of the field to include the non-numeric data. If it turns out that the invalid data is actually a part of the field when it is first populated, you should check the logic in the program that first creates the content (of a record or database).

If you are unable to correct the logic that creates the initial value, you can use code, such as the following, to change invalid data (such as spaces or low-values) to a zero. In most cases, this will allow your program to continue to create final output that is compatible with the V4.2 version of your program.

```
05  NumField  Pic S9(3)V99.
  . . .
If NumField Numeric
    Continue
Else
    If NumField (2:1) = Spaces or Low-Values
        Move Zero to NumField (2:1)
    Else
       Perform Process-Bad-Data
    End-If
End-If
```

Random bad data

After you have identified and corrected all the situations listed above, you may determine that there are one or more fields that occasionally (but not always) contain invalid data. This may (or may not) even include EBCDIC character strings. In most cases this indicates that you have a problem with your program logic. This problem may be in the current program or in the program that first creates the data. You will need to resolve the problem on a case by case basis. It is highly likely that you will not be able to exactly duplicate the V4.2 output with your V6 program. On the other hand, in such a case, it is likely that your V4.2 program was not creating totally correct output when the program logic had an issue.

If you are unable to determine where and how the invalid data was created or populated, you will need to add an explicit numeric class test and (after reporting on the bad data) simply skip this record.

Moving to Production

The ease and success of your moving of the V6 object code into production will depend on the extent to which you were able to test the modified code. If, as suggested above, you were able to run your V4.2 and V6 code in parallel or both in a testing environment with full sets of production data then your migration of the V6 code to production is likely to be easy and successful. This is why it is so important to have source code that compiles cleanly with both versions of the compiler.

If you were unable to run sufficient tests of your V6 code with full production code, then it is suggested that you compile it with ZoneCheck(Abd) for the first few runs in production. This may not be possible if your program is highly performance sensitive. If you are able to do this, then you should have a version of the code compiled with V4.2 and available to your operations team to move into production should a production ZoneCheck ABEND occur with the V6 code.

Remember, that your ultimate aim for the V6 production code is to have a version compiled with

```
FLAG(I,I),Opt(2),NumProc(PFD),NoSSRANGE,NoZoneCheck
```

Once you have production code compiled with these options and V6, you should see comparable or often better production performance with the expected run-time results.

Related Tasks

Although cases of invalid data that can be detected with ZoneCheck are the most common inhibitors to a successful migration to V6, there are several other tasks that it is suggested that you pursue before moving a specific V6 program into production. Obviously these are in addition to all the normal testing that you do for production code.

- Compile with Flag(I,I),DiagTrunc,Rules(NoEvenPack) and resolve all informational and warning messages.
- Compile with SSRANGE(NOZLEN) and test with full sets of production data to determine if you have any invalid subscripting, indexing, or reference modification.
- The ZoneCheck compiler option can help you find cases where there is a mismatch between the Picture clause having or not having a sign for Zoned Decimal data items. However, it does not help you find such cases for internal decimal (Comp-3 or Packed-Decimal) items. If you have previously used the NumProc(NoPFD) compiler option to correct such sign problems, you will need to manually find such mismatched internal decimal items. You can use a similar code snippet as above (for Zoned Decimal items) for internal decimal items – once you have discovered when and if you have such problems.

- Use the "Scan for Compatibility" feature of RDz or manually scan your programs to determine if the parameters in your calling and called programs match.