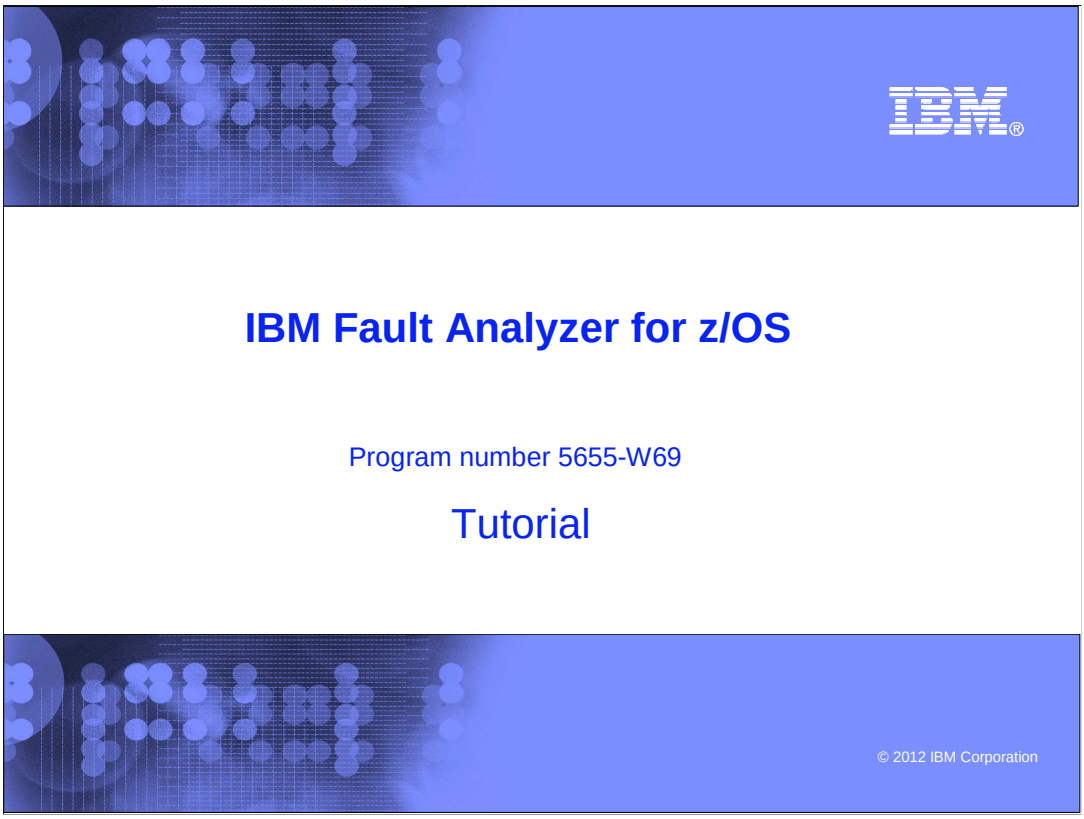
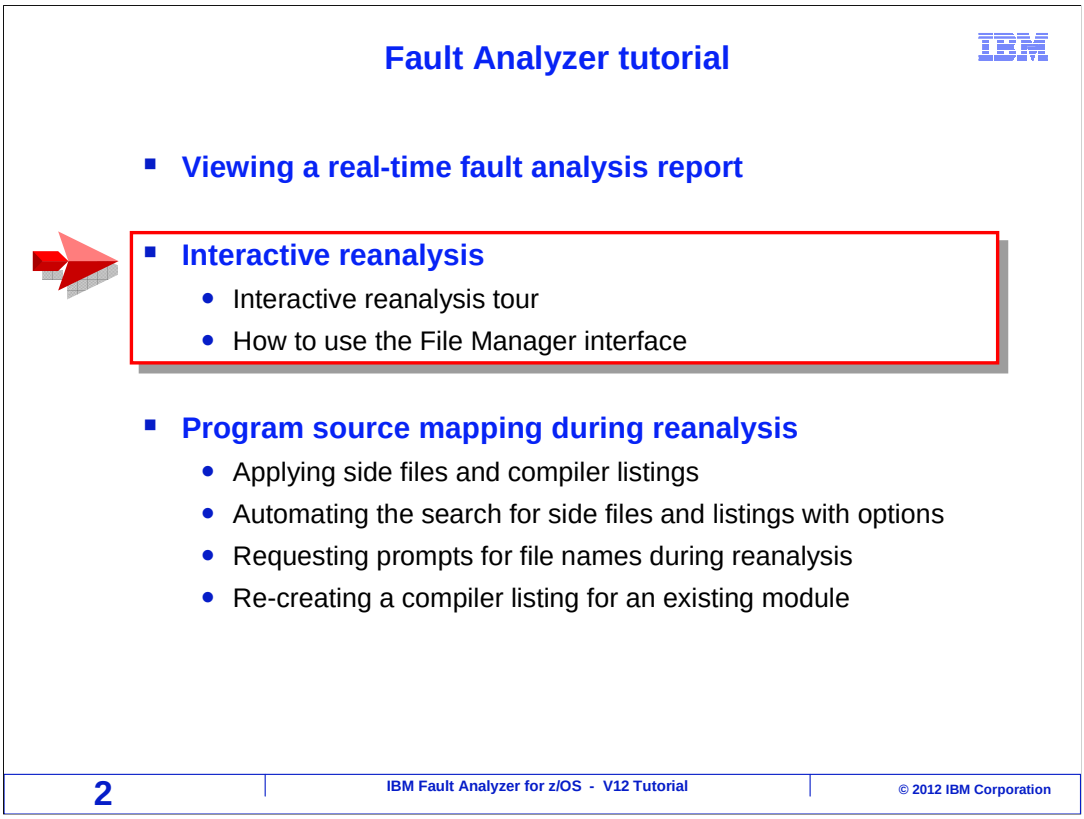


This is the tutorial for IBM's Fault Analyzer for z/OS®, one of the IBM zSeries® problem determination tools.



In this section you will see how to use the interactive reanalysis function. You will see how to start your interactive reanalysis session, and you will see an example of performing root cause analysis.

Why do reanalysis?

- Interactive reanalysis gives you easy navigation to abend and program information source
- If the abending program's source listing was not available to Fault Analyzer when the abend occurred
  - In the Real-Time Report, you got:

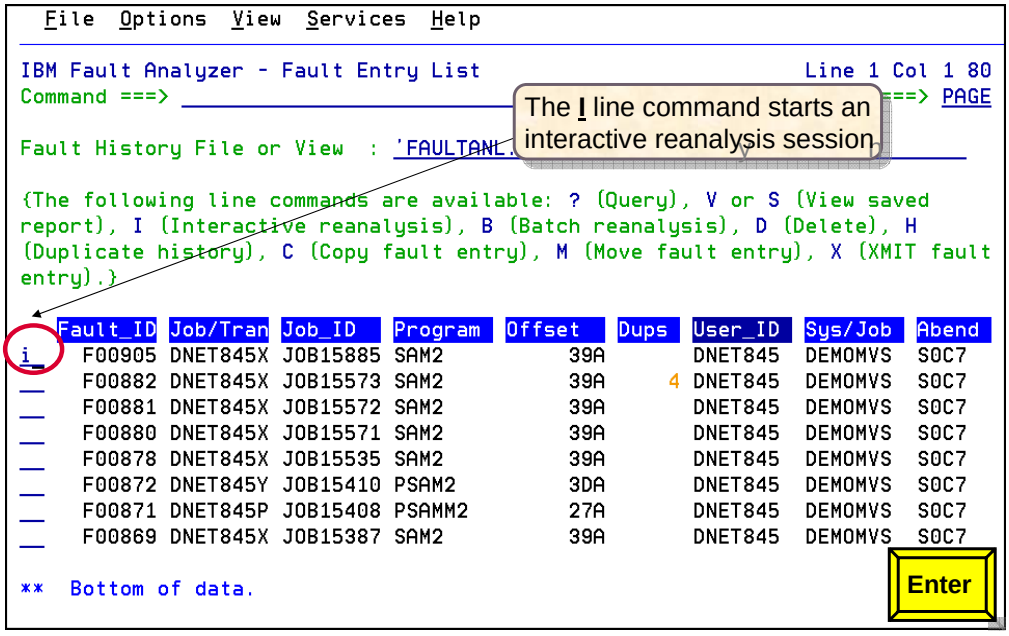
```
A system abend 0C7 occurred in module IDISCBL1
CSECT IDISCBL1 at offset X'3D0'.
The abend was caused by machine instruction FD73D0B8D0A8(DP)
```

- You can apply a side file or compiler listing
  - Then with Reanalysis, you can get:

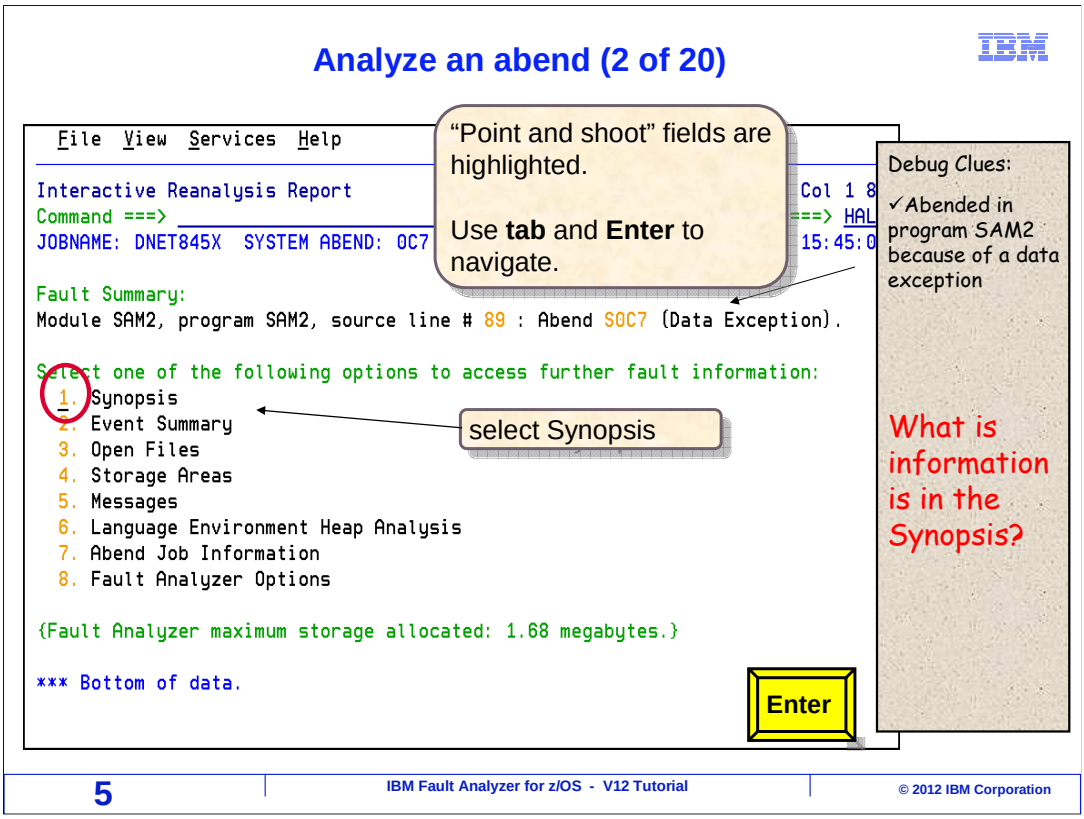
```
The cause of the failure was program IDISCBL1.
The COBOL source code that immediately preceded the failure was:
Source
Line #
------
000051     DIVIDE NUMBERX BY ERROR_COUNT GIVING BAD_RESULT
```


When an abend occurs, fault analyzer automatically produces a real time report. So the question arises, why do a RE-analysis? Interactive reanalysis gives you much easier navigation to get to the information you are looking for compared to using a real time report. In addition, if source information was not available to fault analyzer when the abend occurred, it might report information like you see in the first box, where it shows the offset of the abend and machine instruction. But to become more productive you want information more like what you see in the second box, where it gives you source level information, including the abending source statement and the values of program variables. When you use interactive reanalysis, you can apply side files and compiler listings after the fact; after an abend has occurred, without having to re-create the abend.

Analyze an abend (1 of 20)



From the Fault Entry List, start interactive reanalysis by typing an I line command next to the abend you want to analyze. Press enter.



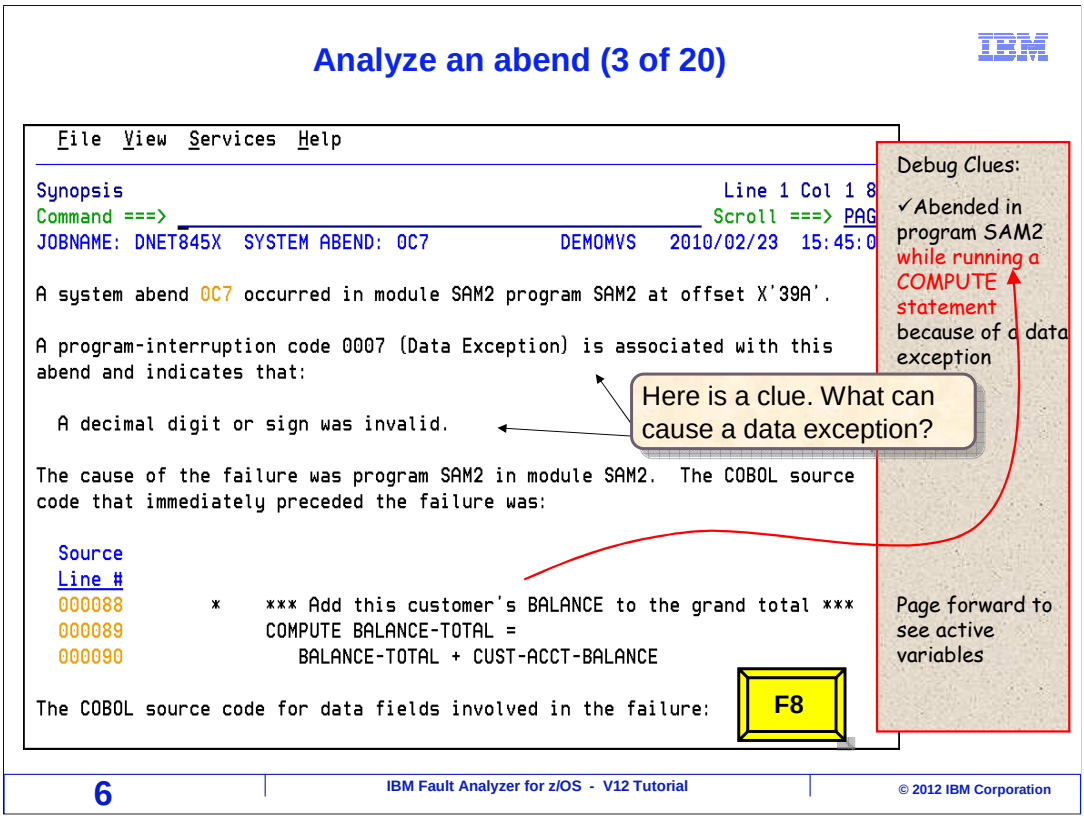
The interactive reanalysis menu is displayed. To research the root cause of a problem, you will make observations about the abend, come up with theories about what happened, and then test those theories. Already on this first screen there is some useful information. You see that the application abended in a program named SAM2, and that there was a data exception.

As this example is researched, a few notes will be shown on the right side of the screen. Here it was noted that the abend occurred in program SAM2 because of a data exception. In this section of the tutorial, you will be given questions, you will have a few moments to consider it, and you can pause the tutorial if you would like more time.

Notice is that there are some yellow highlighted fields. These are called point and shoot fields. If you use the tab key on your keyboard, the cursor will jump right to those yellow highlighted fields. When your cursor is on one of them and you press enter, It jumps to the screen indicated by the selection.

This is the main menu for interactive reanalysis, and options are: synopsis, event summary, and others. These are the same sections that are shown in real-time reports, but the menu gives you an easier way to get directly to a section you want to see. Here is the first question, and it is a review from a previous section, "viewing a real-time report". “What information does the synopsis section show?” Pause the tutorial now if you would like more time.

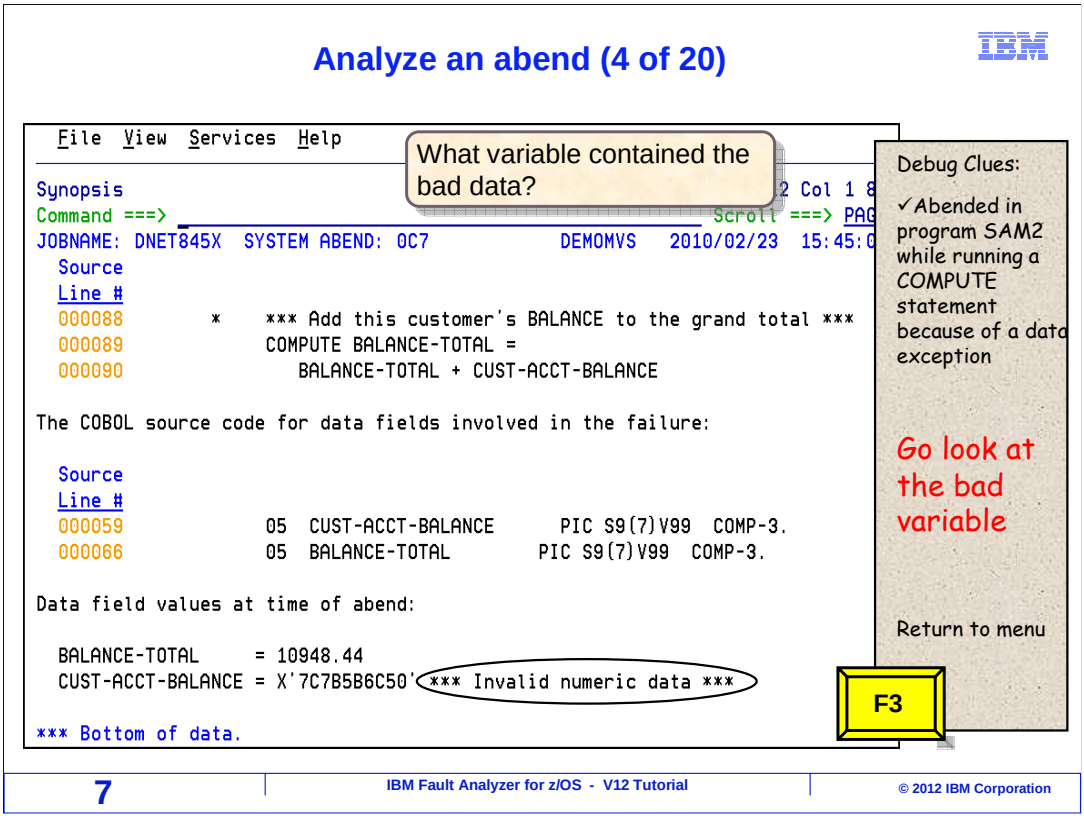
The synopsis is a good place to start your research. It contains an explanation of the abend code, and when available shows the program statement or machine instruction that caused the abend. To see it, tab to the point-and-shoot field for the synopsis section, and press enter.



Now the synopsis is displayed. You see that the abend occurred in a program called SAM2. Also notice that the source statement that caused the abend is shown - it was a COMPUTE statement.

Here is another question. What can happen in a program that can cause a data exception?

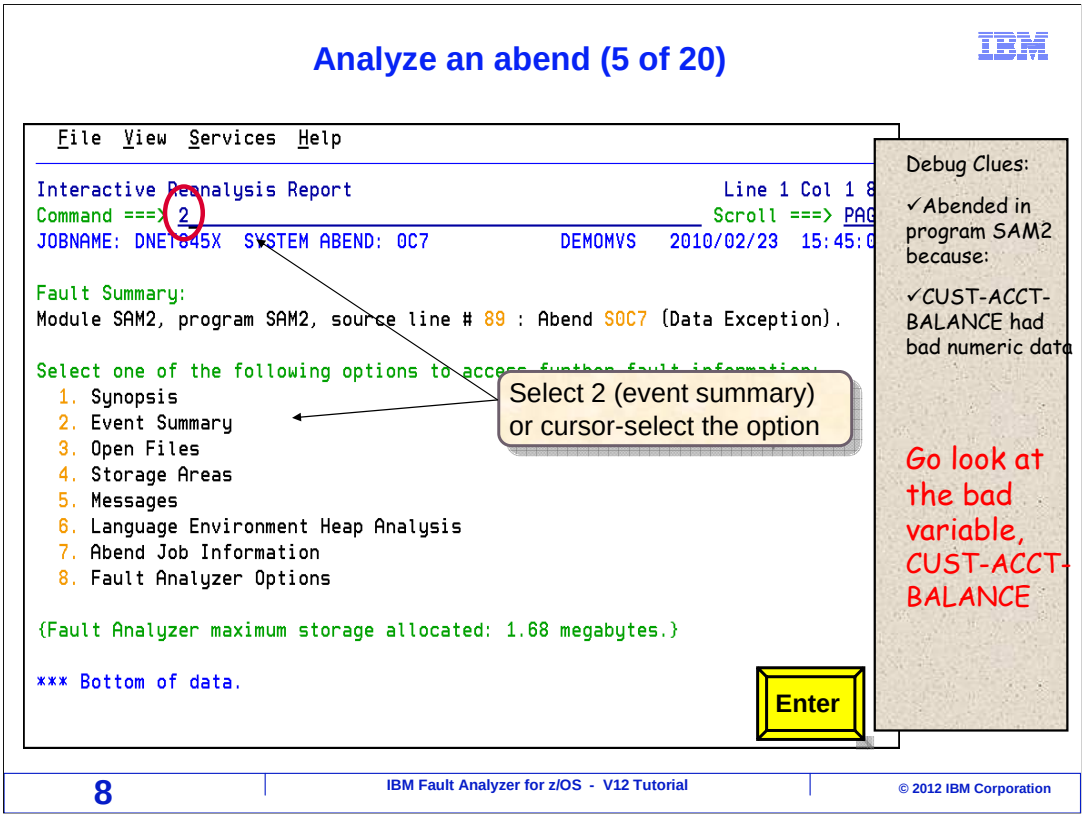
Fault analyzer is showing the description: “A decimal digit or sign was invalid”. A data exception is caused when a program attempts an arithmetic operation, but one of the data elements contains non-numeric data. Scrolling down further in the synopsis, F8.



You can see the COMPUTE statement that abended. Here is the next question: What variable contained the bad data?

Near the bottom, fault analyzer flagged variable CUST-ACCT-BALANCE. Notice that it is defined as a comp-3 field, which is packed decimal. If you are familiar with packed decimal fields, you can see from the hexadecimal representation that it had bad data.

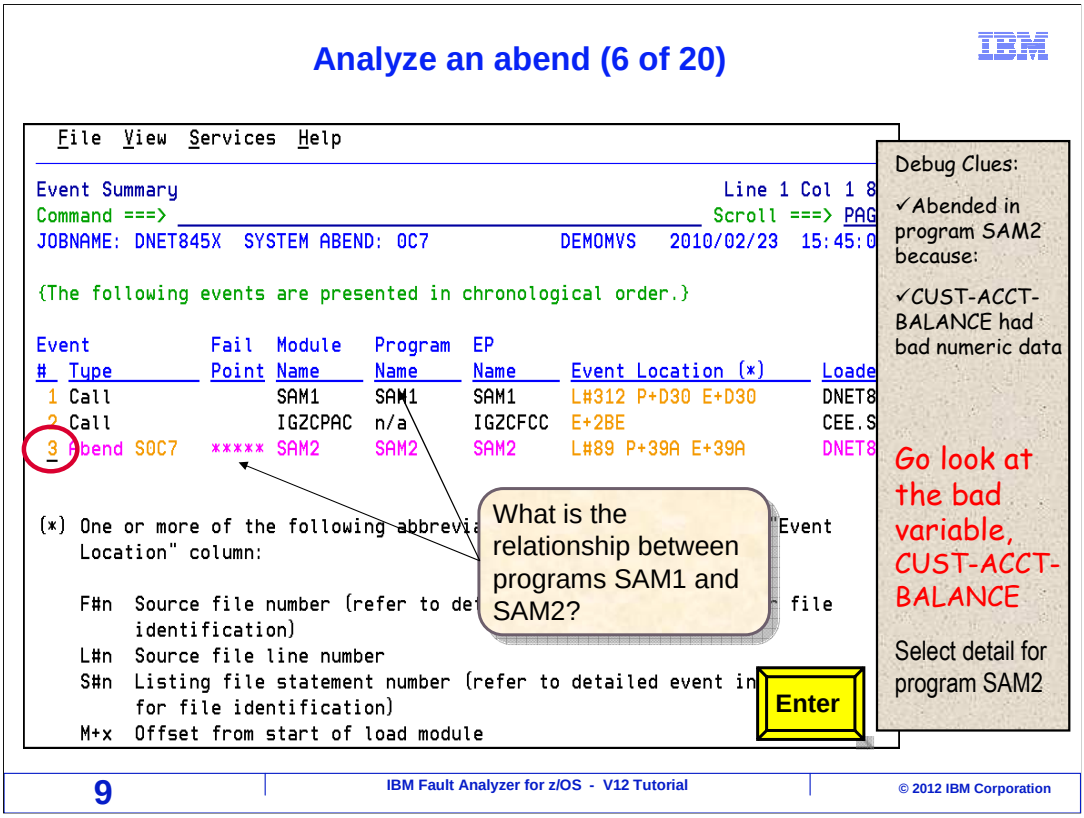
Now the local cause of the abend is known, although not necessarily yet the root cause. The goal of this exercise is to understand the problem in enough detail that it can actually be fixed, and the program can run again without an abend. More research is needed to find the root cause. F3 returns to the menu.



Consider the strategy you might use to research an abend. There are a lot of different ways to go about it.

In this example, you already know that the abend occurred in program SAM2, and that the abend was caused because of bad data in a variable called CUST-ACCT-BALANCE. From here you could go a lot of directions to get more information. For example you could look at the source listing to understand the logic of the program to try to understand where the bad data came from. In this case since there is bad data in a variable, you will see a “follow the data” strategy used, tracking the bad data back to it’s origin.

The next step in this example is to look at the definition of the bad variable CUST-ACCT-BALANCE in SAM2, to see if other clues can be found there that will help with the analysis. One way to get there is through the event summary. You can either tab to the Event Summary point-and-shoot field, or specify option 2 on the command line.

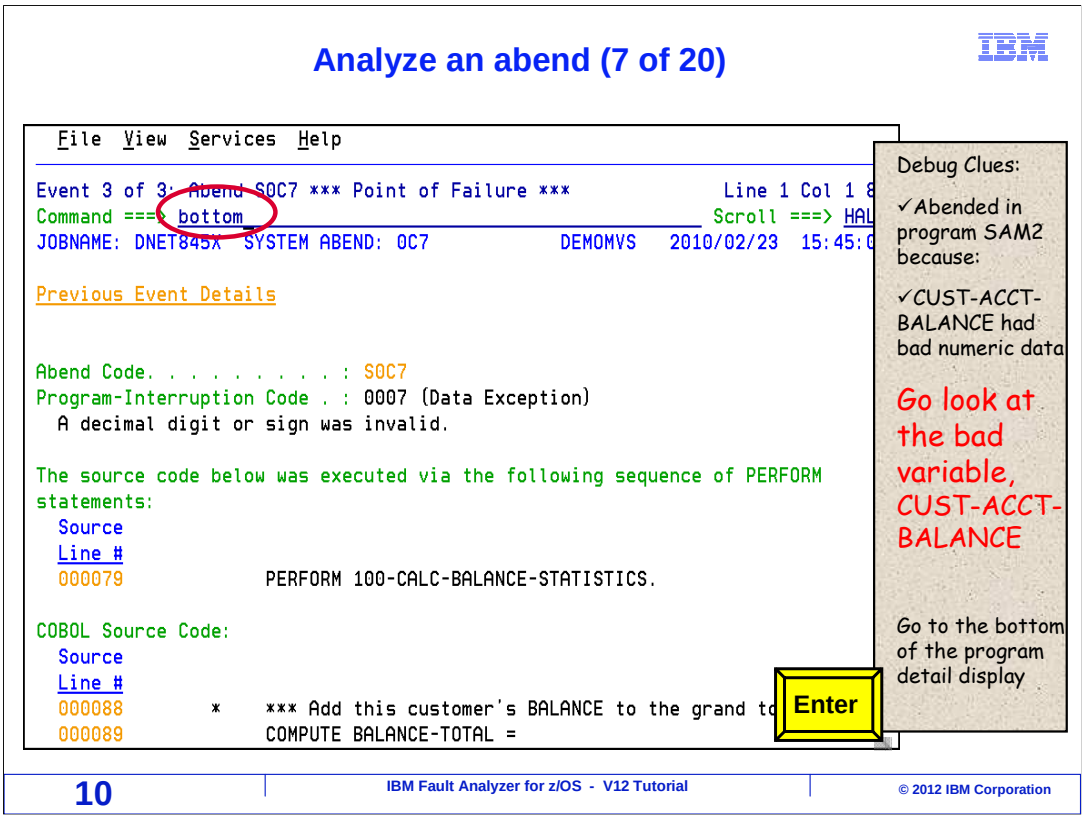


The event summary shows the program call chain. The next question is: “Based on what is shown, what is the relationship between programs SAM1 and SAM2?”. Pause the tutorial now if you would like more time.

The program at the top of the list is the first program on the call chain. Typically, this is the main program. In this case, SAM1 is the main program, and SAM2 is a subroutine that was called by SAM1. You may be wondering about the IGZ module in the middle. That is a system routine that serviced the call request. You may often see system modules in the call chain, and very often, as in this case, you can ignore them.

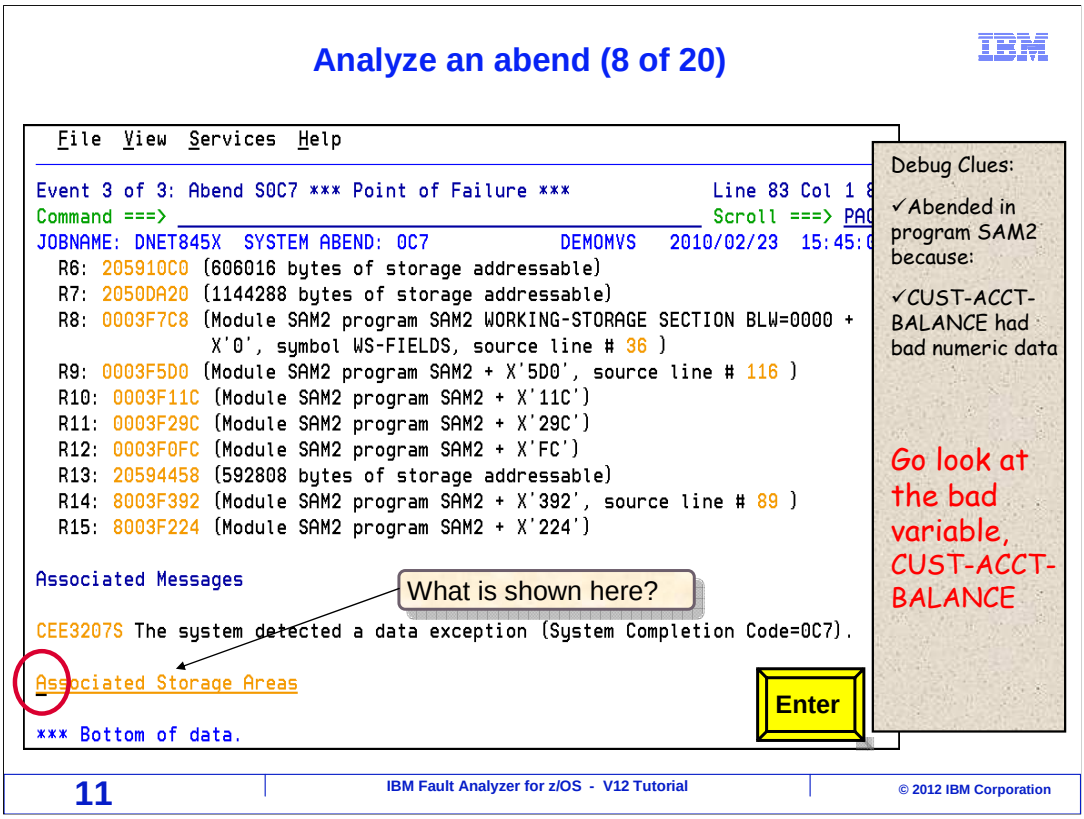
The variable of interest is in SAM2, where the abend occurred. So tab down to event number 3, which is program SAM2, and press enter.

Analyze an abend (7 of 20)



Now the detail report for event 3 is displayed. Again, the next step is to see variable CUST-ACCT-BALANCE and how it is defined. So go to the bottom of this section. One way to get there is with the “bottom” command.

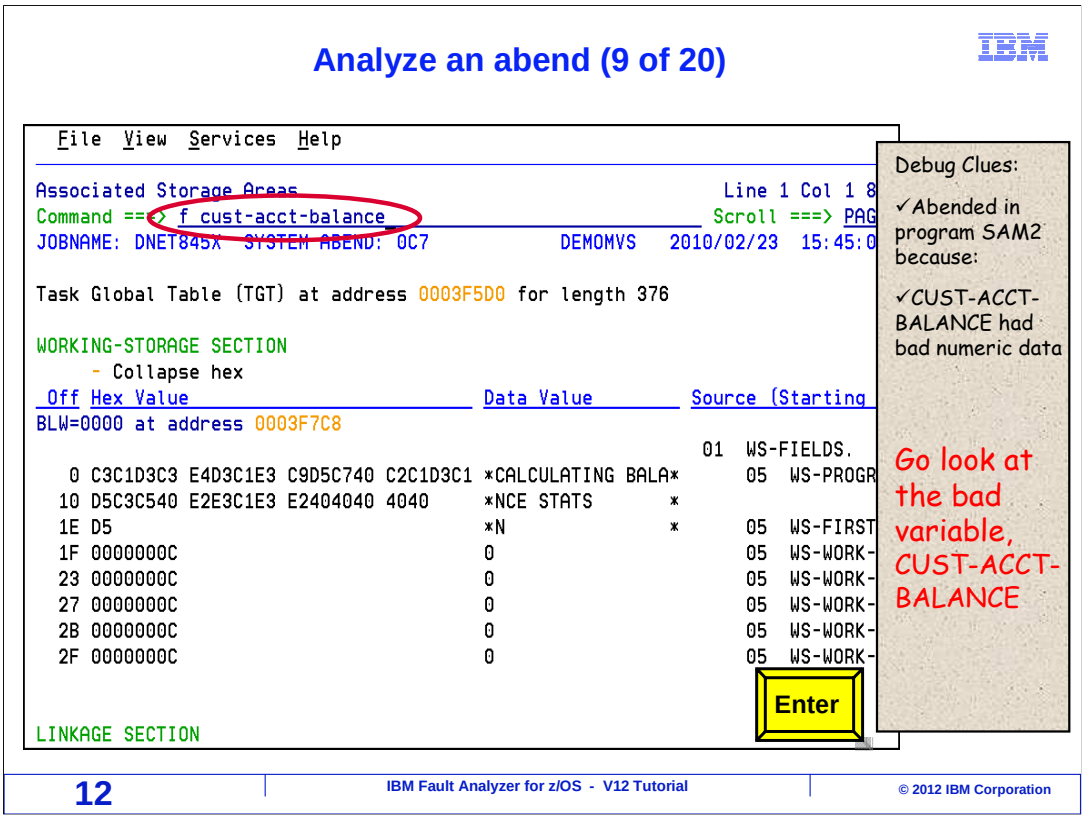
Analyze an abend (8 of 20)



The cursor is tabbed to the point-and-shoot field for “associated storage areas”. Here is the next question: “What information is shown in associated storage areas?”

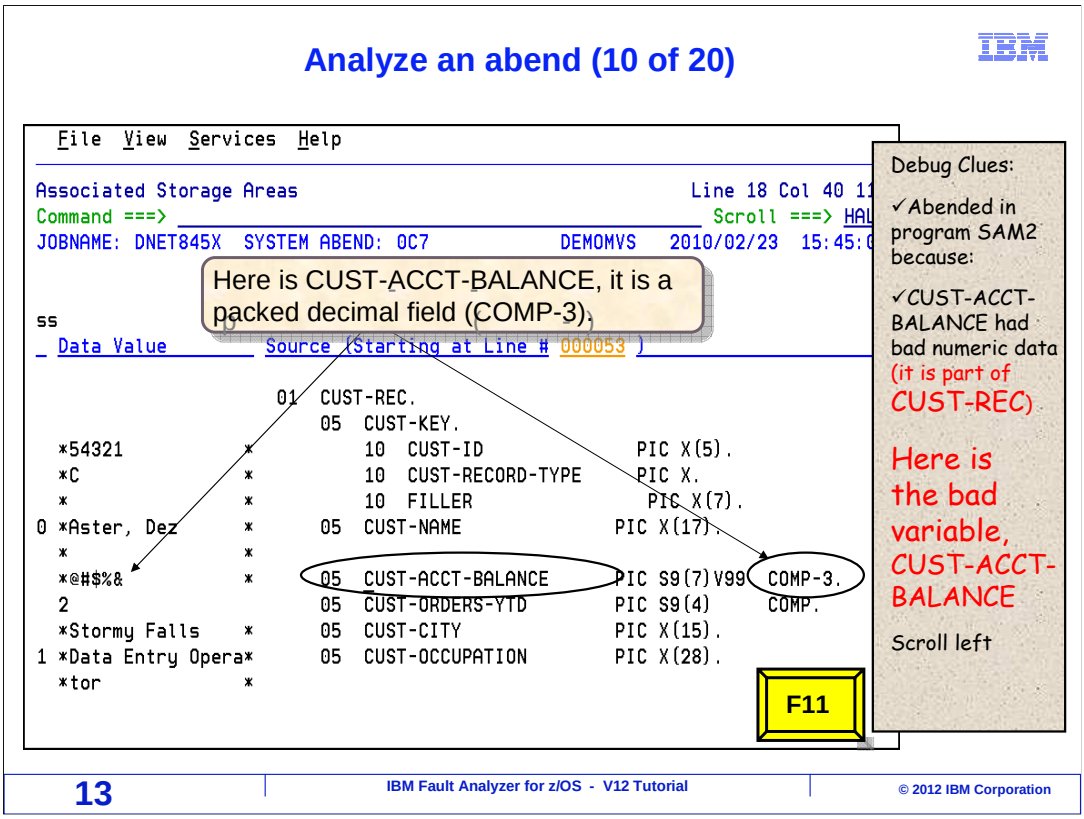
Pressing Enter…

Analyze an abend (9 of 20)



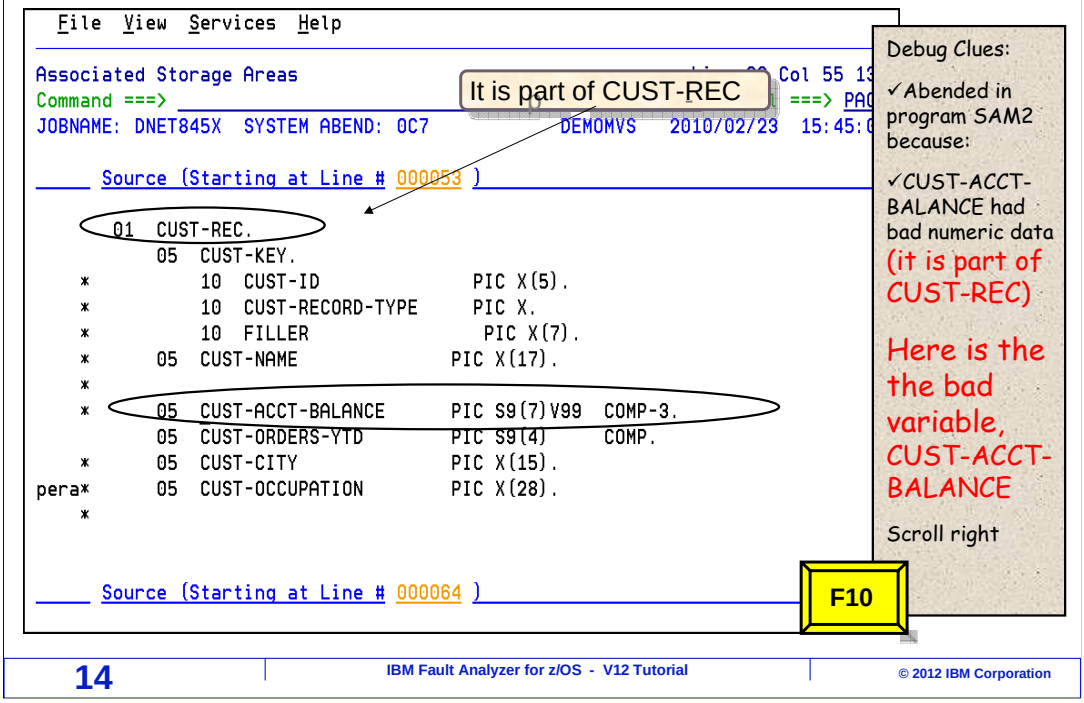
Associated storage areas shows the program's variables, and possibly other areas if the program explicitly allocated storage. An easy way to locate a variable is with a Find command. Type in F, space, and the variable name on the command line, and press enter.

Analyze an abend (10 of 20)

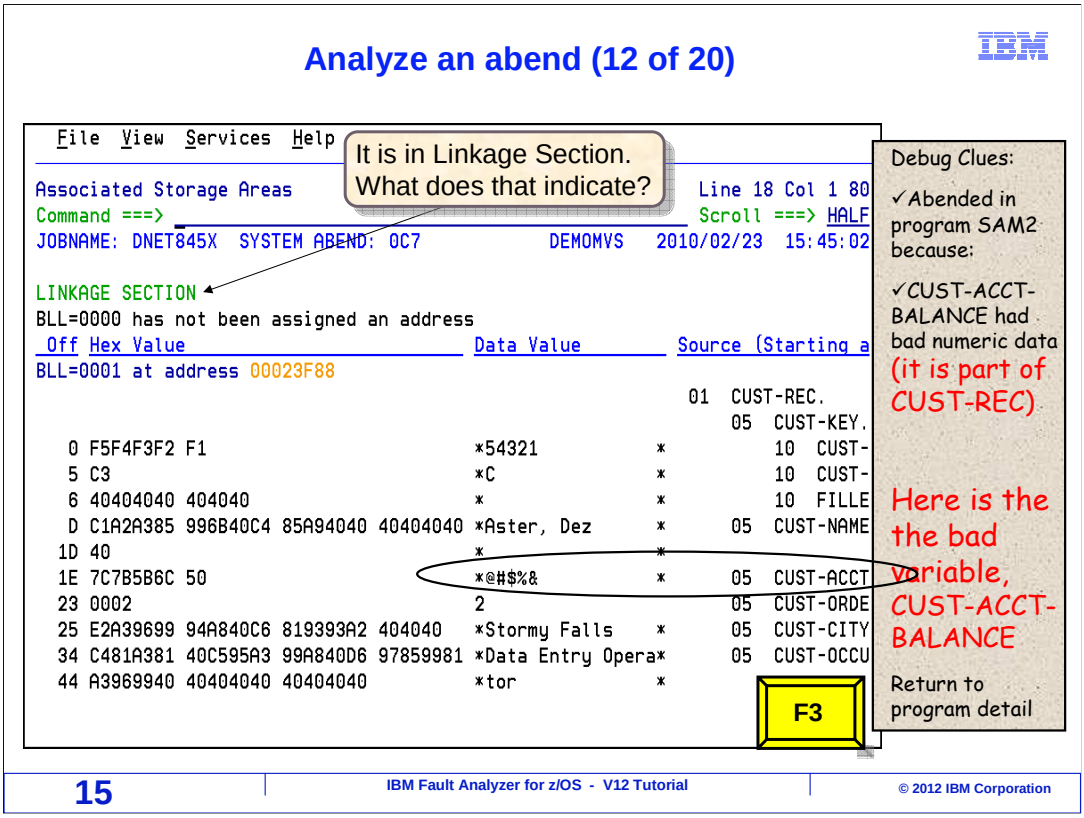


Here is the variable. Notice that it contains the bad data. It does not look very numeric. Scrolling over to the right, F11.

Analyze an abend (11 of 20)



Notice that the bad variable is within a data structure called CUST-REC. Scroll left to see more information.



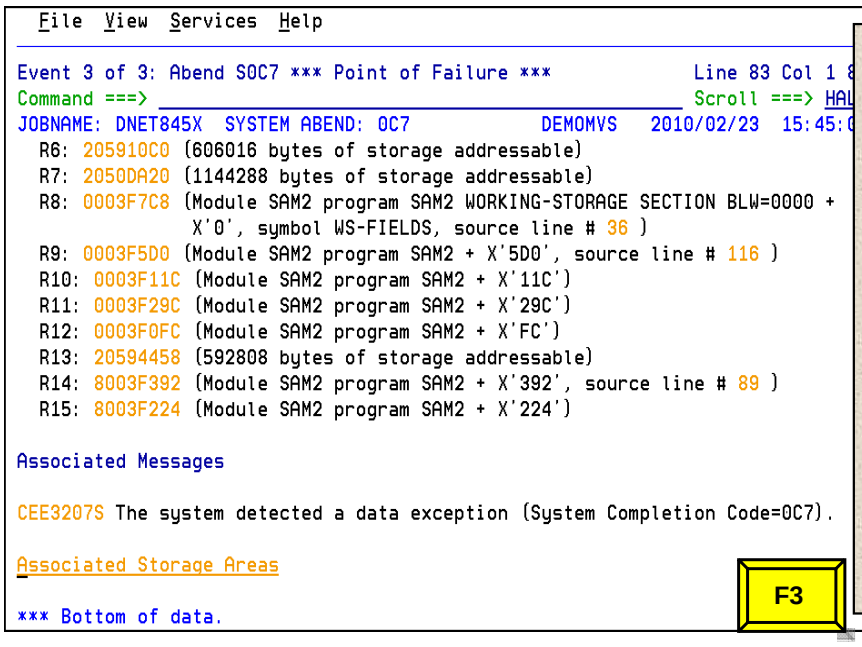
Notice that the bad variable is in linkage section. And here is the next question: “Since it is in linkage section, what can be said about where the data came from?”. A little experience with COBOL is needed to answer this one.

In COBOL, data is passed between a calling program and a called program in Linkage section. So the bad data in CUST-ACCT-BALANCE may have been passed from the main program.

Part of troubleshooting an abend involves developing theories about what may have caused it, and then testing those theories. So now there is a theory - the bad data may have been passed to this program from the main program. The bad variable is in a data structure called CUST-REC. COBOL programs pass data at the 01 data structure level. Also notice that CUST-REC is the first item in linkage, which means it is the first parameter in the list passed by the calling program. That will be helpful information in just a minute.

To test the theory, the next step is to go back to the main program and see what data it passed. F3 to exit.

Analyze an abend (13 of 20)



Debug Clues:

✓Abended in program SAM2 because:

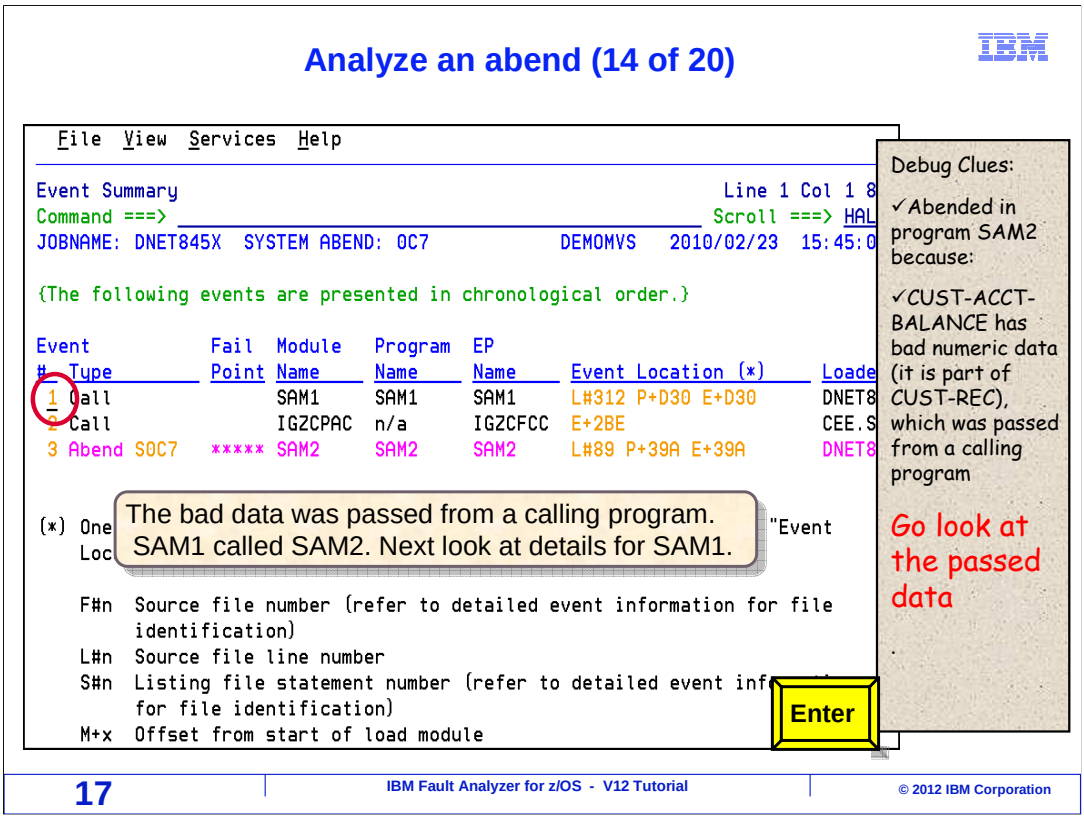
✓CUST-ACCT-BALANCE has bad numeric data (it is part of CUST-REC), which was passed from a calling program

Go look at the passed data

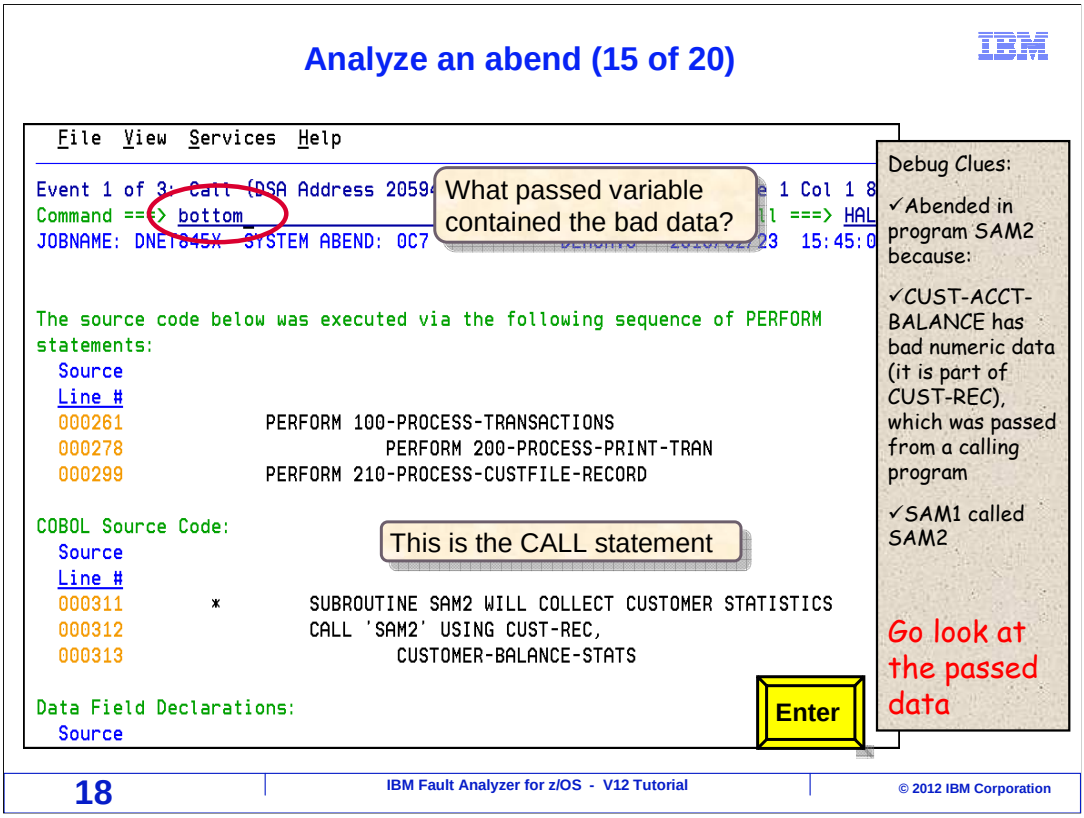
Return to the events list

That returned to the detail report for program SAM2. F3 again…

Analyze an abend (14 of 20)



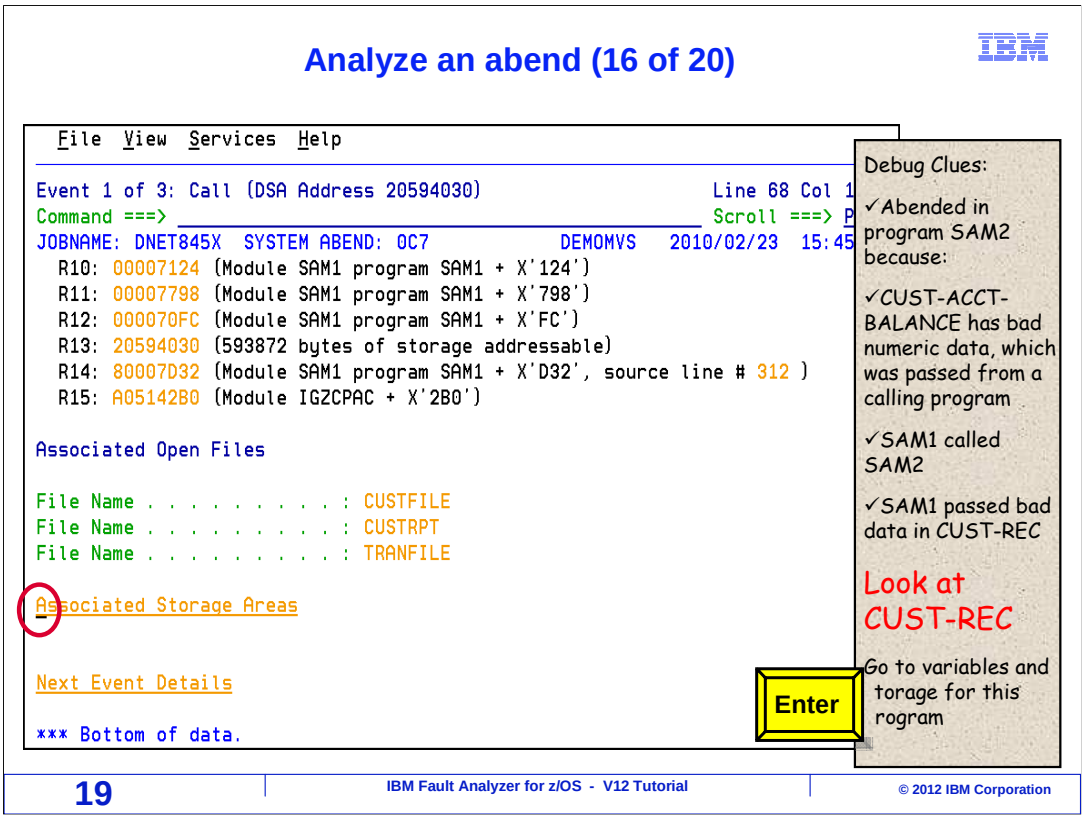
… returns to the event summary. Now, tab to event one, which is the main program, SAM1. Press enter.



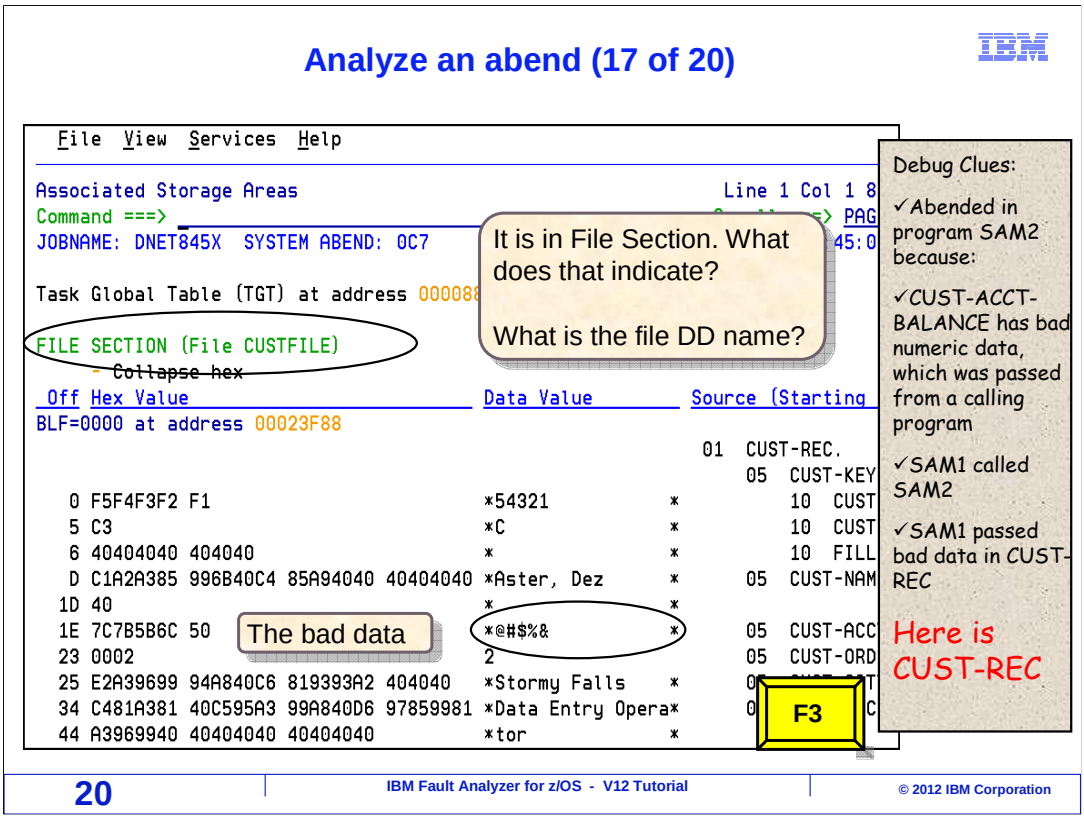
That displays the detail report for the main program. The active statement in the main program is shown; it is the CALL statement for the subroutine. You can see the names of the variables that were passed. Here is the next question, which passed variable might have contained the bad data?

In the subprogram, you just saw that the bad data was in the first passed parameter. So, you know that variable CUST-REC is the one of interest, because it is the first variable passed by the CALL statement. The next step is to look at the data in CUST-REC, to verify that it has the bad data. One way to see it is to go to the associated storage areas. The “bottom” command is used.

Analyze an abend (16 of 20)



The cursor is tabbed to the point-and-shoot field for “associated storage areas”. Press enter.



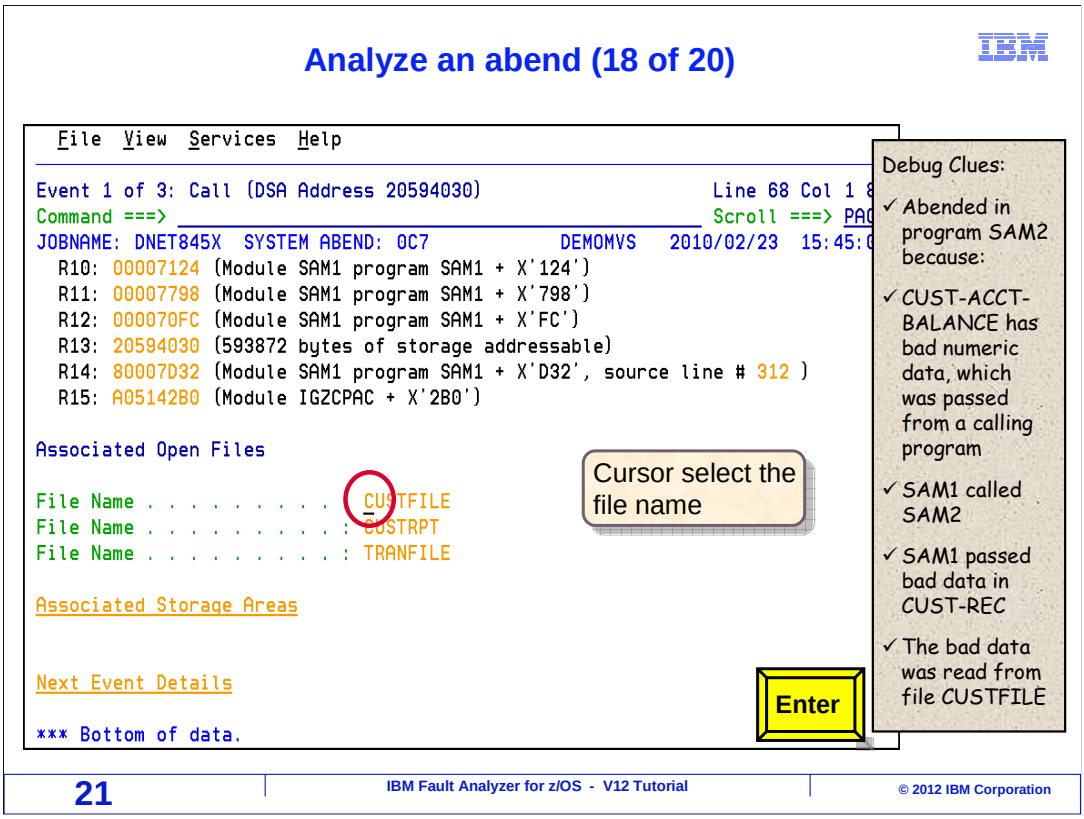
In the associated storage areas, you can see the program variables. CUST-REC happens to be near the top of the section, although if it were not you could use a Find command to locate it quickly. You see that the CUST-REC structure contains CUST-ACCT-BALANCE. Of course, you could scroll to the right to see the rest of the variable name. The same bad data is shown, so that validates that the data has been tracked back to here. So far, the theory that the bad data was passed from the calling program still makes sense.

There is another clue on this screen about where the data may have come from. And here is the next question. “This data is in File Section. What does that mean about where the data may have come from?”.

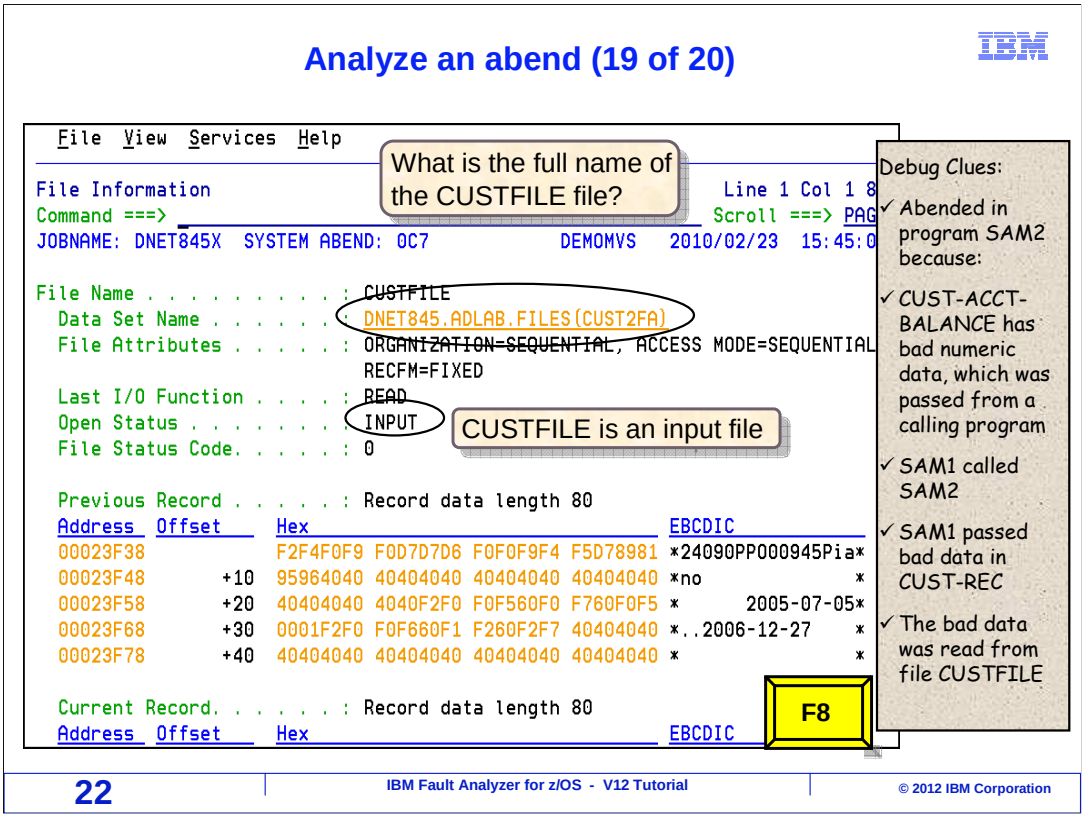
File Section has variables that are read from or written to records in files. Since the bad data is in File section, another reasonable theory is that the data may have been read from a file. The DD name is CUSTFILE.

In the next step the file buffers are examined, to see if the bad data was in a record that was read from the file. F3…

Analyze an abend (18 of 20)



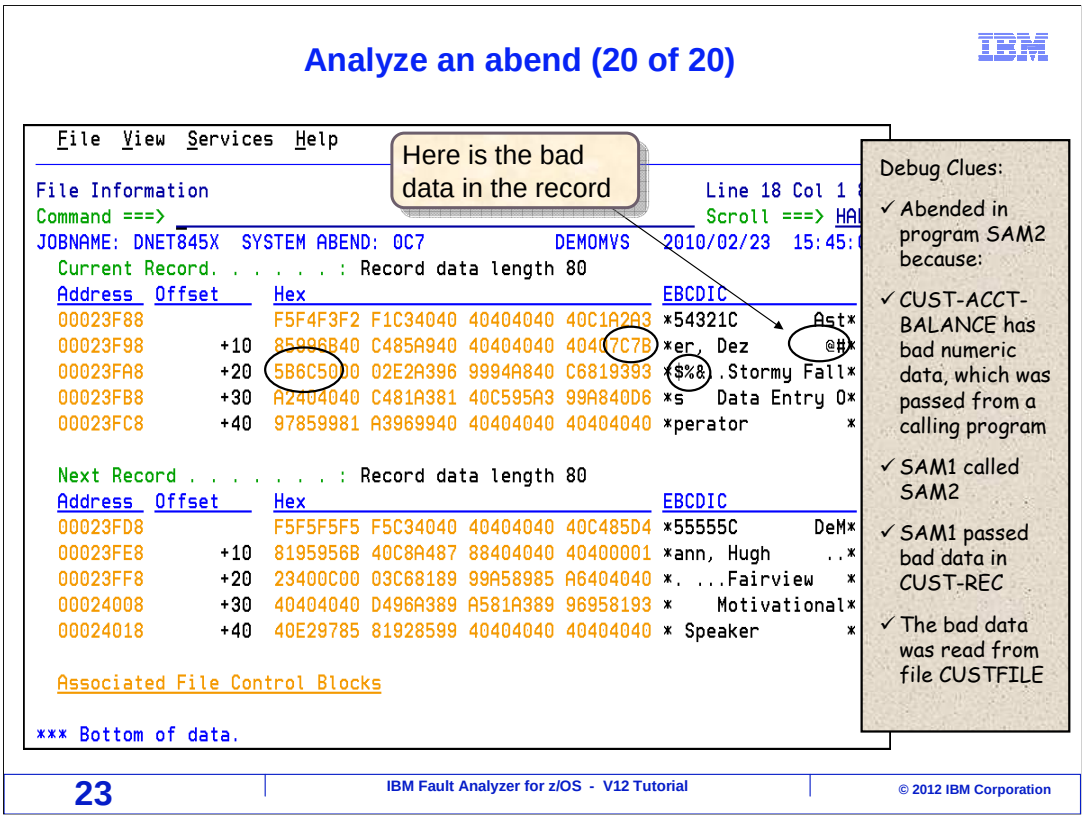
…returns to the detail report for program SAM1. Here is a list of files that were open at the time of the abend. You can tab down to the point-and-shoot field for CUSTFILE. Enter.



And the File Information panel for CUSTFILE is displayed. Here is the last question: “What is the file name for the CUSTFILE DD?”.

The file name is shown in the data set name field, and it is DNET845.ADLAB.FILES(CUST2FA). Notice that the file is opened for input only, and therefore it is likely that data in the record buffer actually came from the file. Scrolling down, F8.

Analyze an abend (20 of 20)

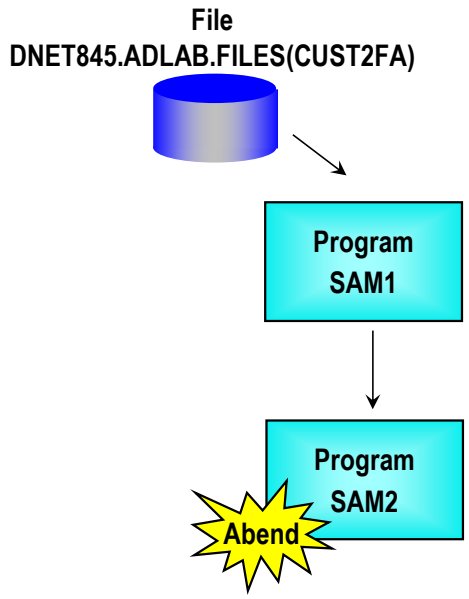


Here is the current record in the file, and sure enough, the bad data is there.

At this point, enough information has been gathered to develop a complete theory for understanding the root cause of this abend.

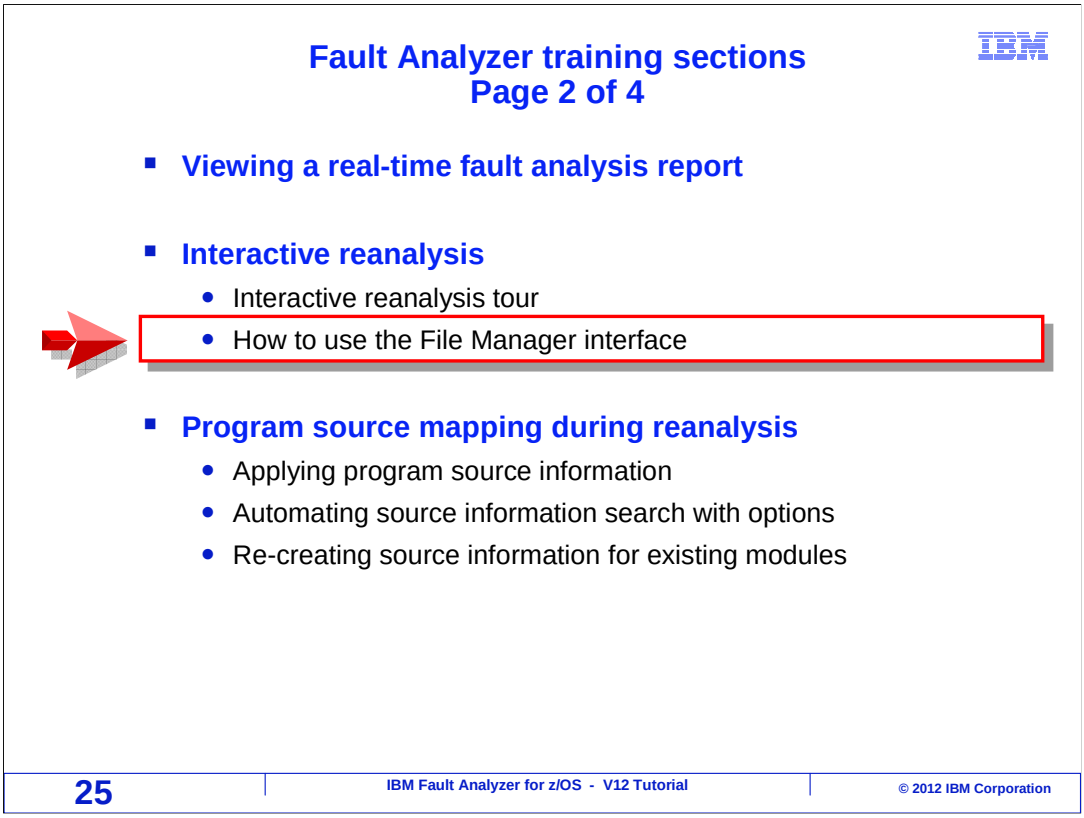
Analyze an abend … what happened:

1. SAM1 read a record from an input file that contained bad data in the CUST-ACCT-BALANCE field
2. Program SAM1 called SAM2, passing the bad data
3. SAM2 abended when it tried to calculate using the bad data



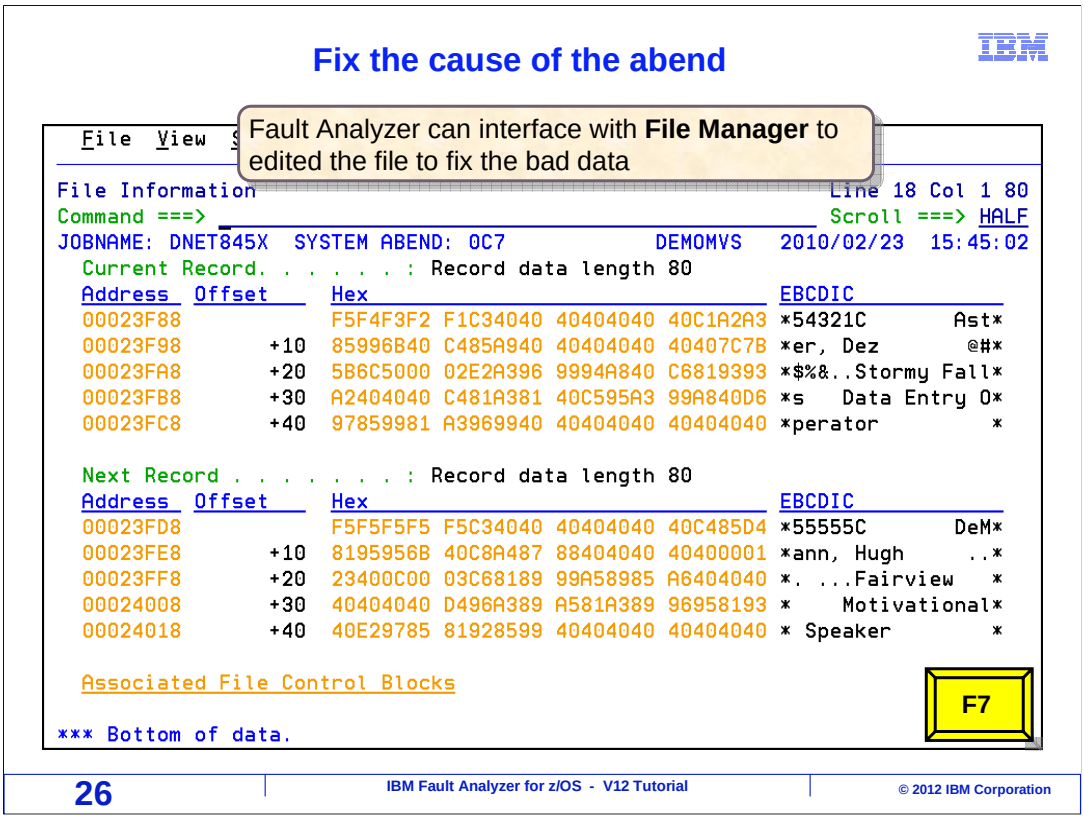
SAM1 is the main program. It read a record from a file. The record contained bad data in the CUST-ACCT-BALANCE field. SAM1 passed the record area with the bad data to a subprogram named SAM2. Finally, SAM2 tried to do arithmetic with the bad data, and that caused the abend.

The example you just saw is not atypical of the process you might work through to troubleshoot an abend. You will look at the symptoms, develop theories about what may have caused the problem, and examine the available information to test those theories.



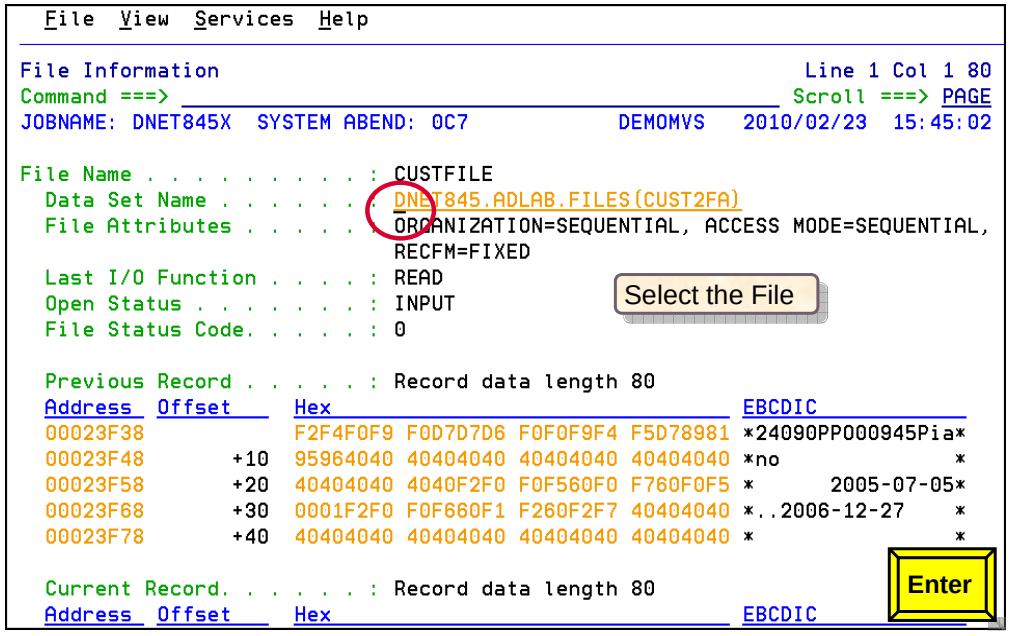
Fault Analyzer can be set up to interface with IBM File Manager for z/OS. Next you will see how to access File Manager to browse or edit files.

Fix the cause of the abend



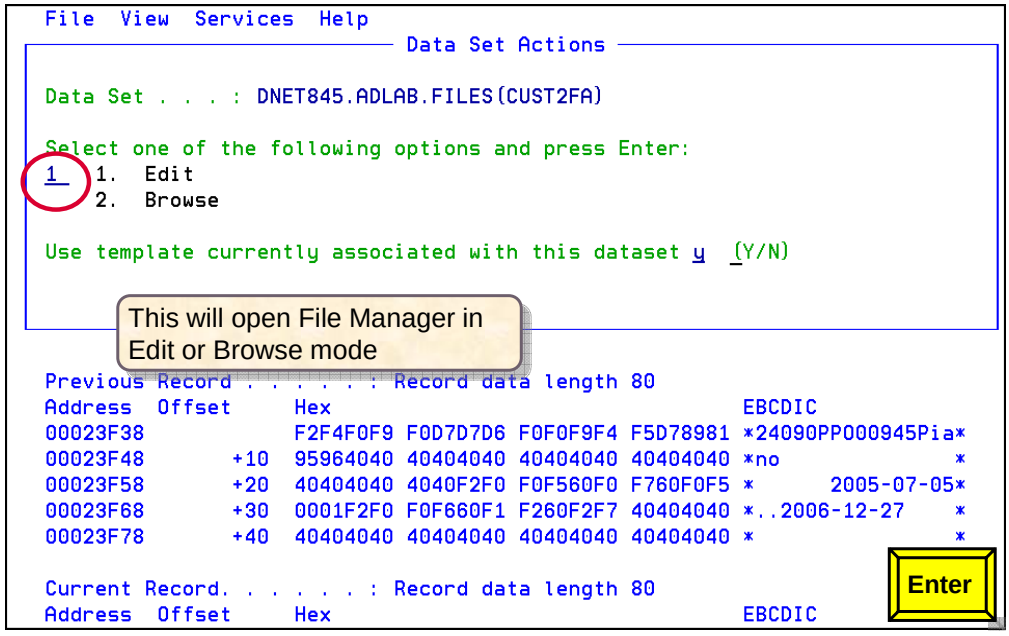
To use this interface, IBM File Manager for z/OS has to be installed on your system, and your systems programmer must have enabled the interface between Fault Analyzer and File Manager. Scroll to the top of the File Information section with F7.

Cursor select the data set name

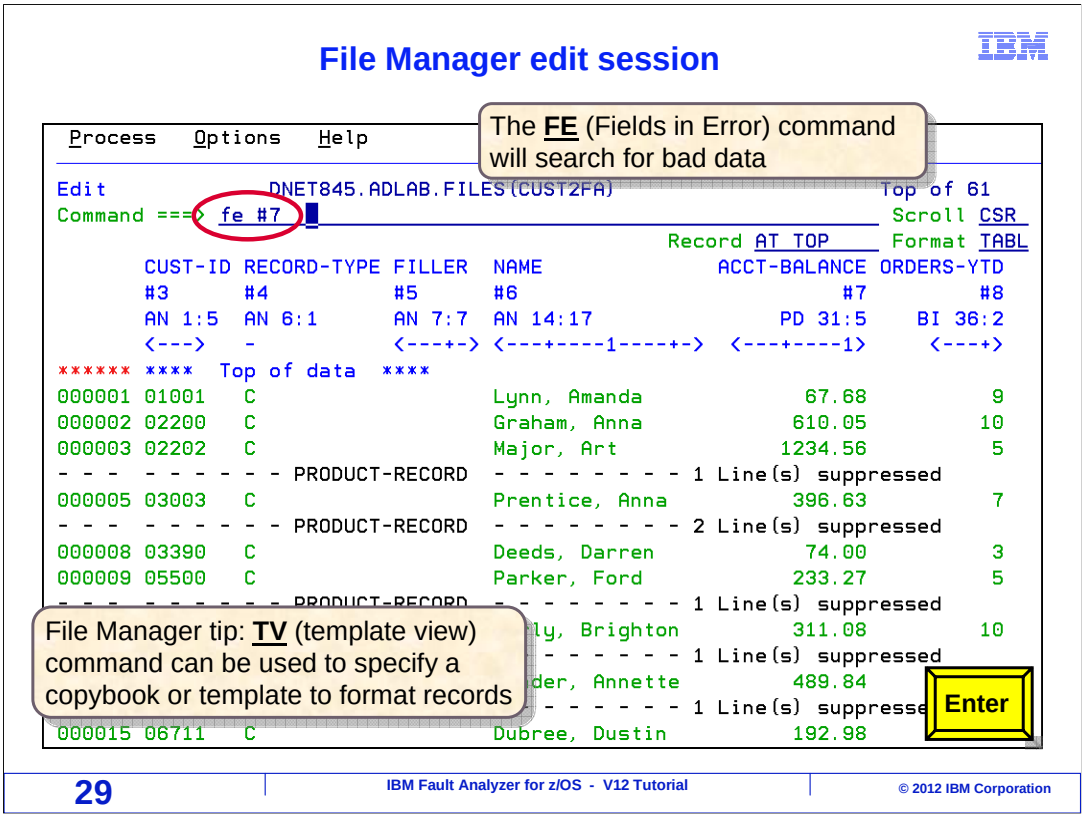


To edit the file and look at the bad data, you have to have authorization to the file. And of course to fix the bad data, it still has to be in the file. But if those are not problems then you can browse or edit the file. Tab to the file name, and press enter.

Select edit

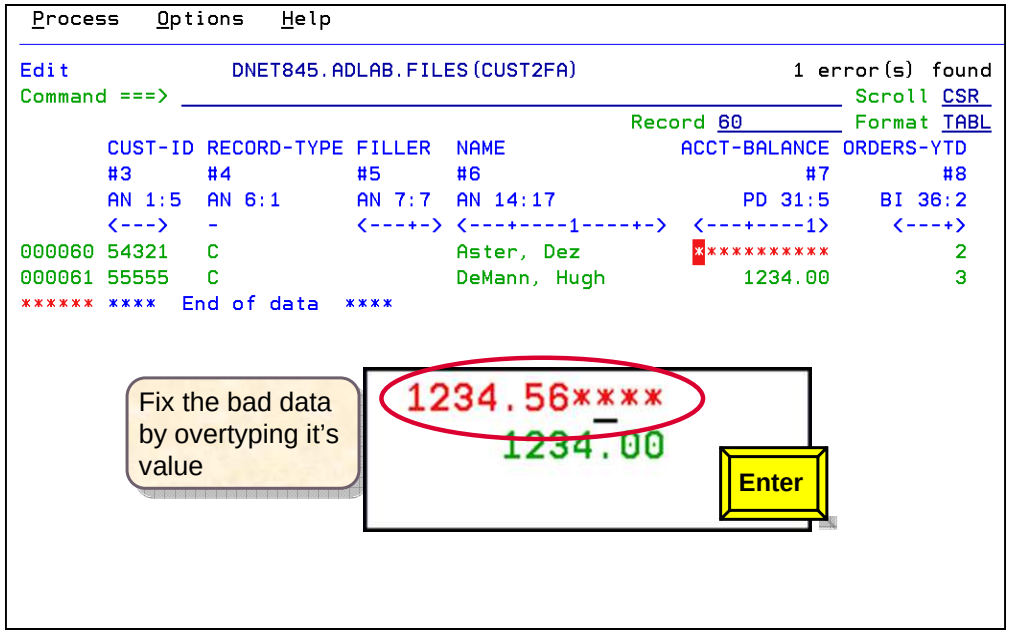


A panel is displayed, where you can choose browse or edit. Option one for edit is entered.



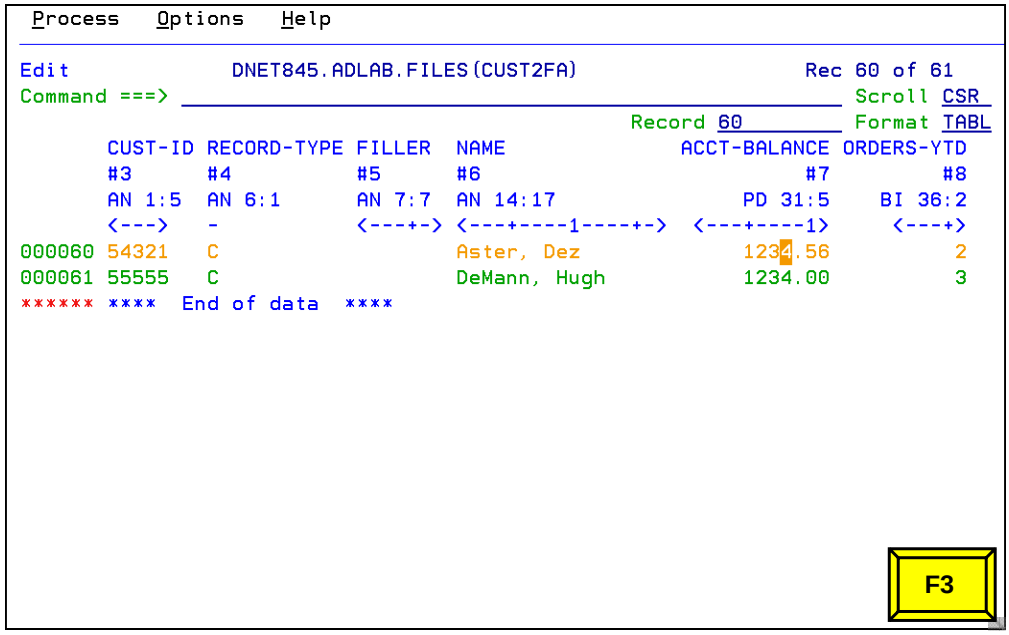
The file manager editor is displayed. The abend was caused because of bad data in a record in the ACCT-BALANCE field. In the File Manager editor, there is an easy way to locate bad data in numeric fields: the FE command, which stands for Fields in Error. It scans records looking at numeric fields and stops at the next one that has bad data.

Fix the bad data



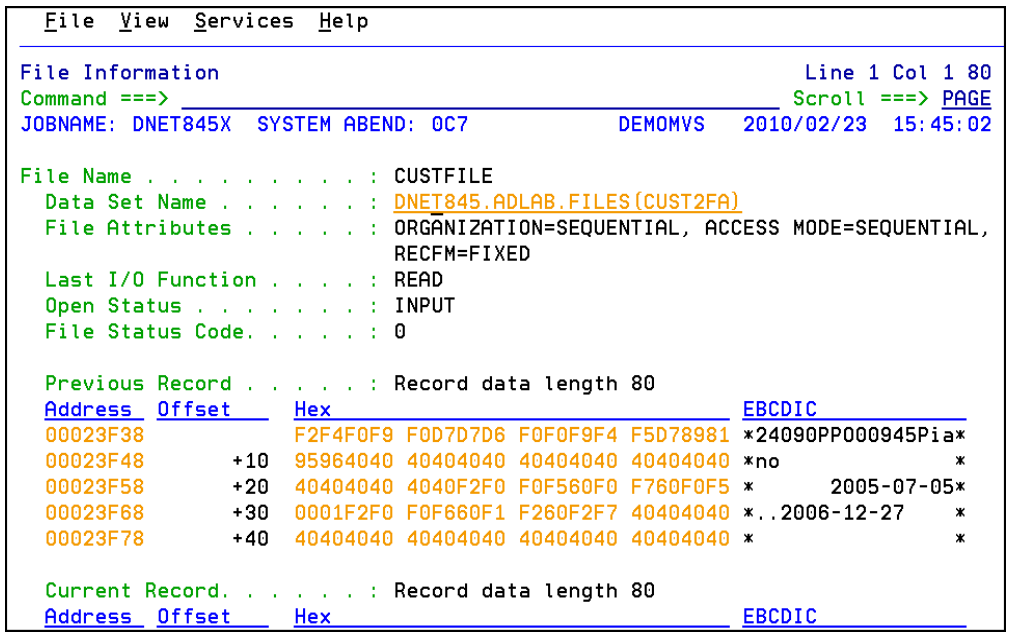
To fix the bad data, overtype the field with a valid value.

Save the change



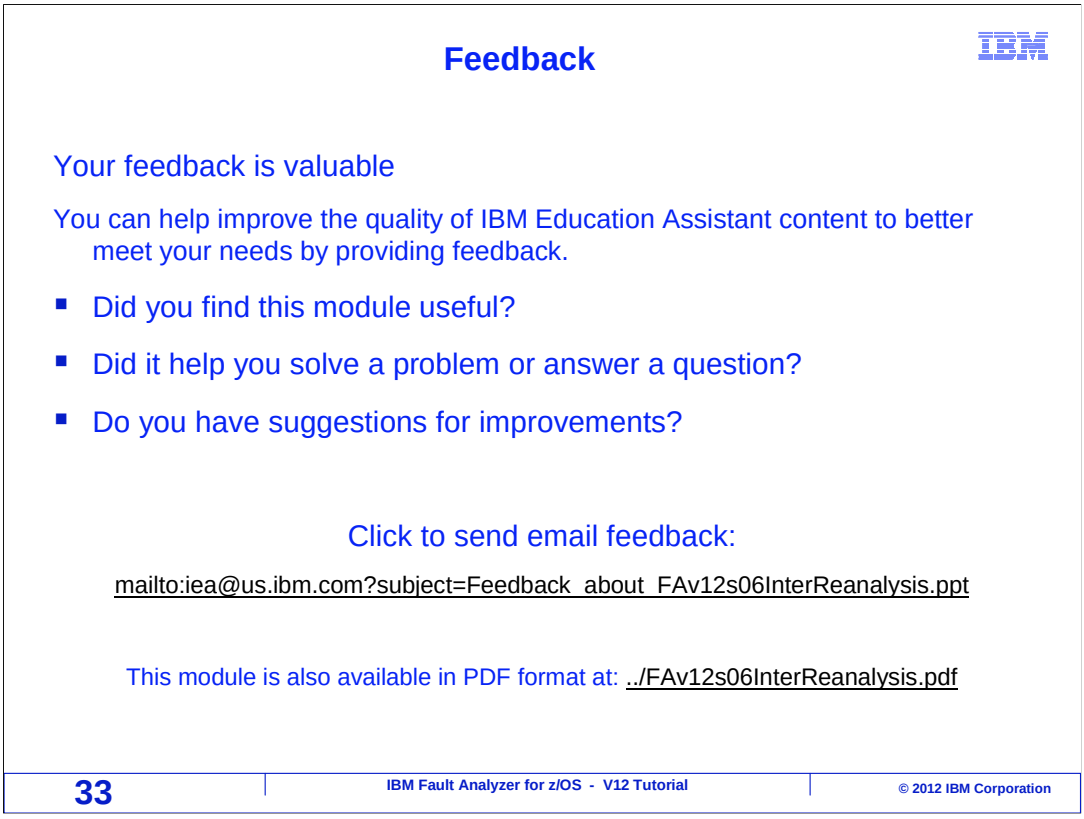
Press F3 to save the change and exit.

File Manager returns to Fault Analyzer



That returns to Fault Analyzer. At this point, the cause of the abend has been fixed. Of course, it is a good programming practice to perform numeric validity checks on data that comes from an external source. Because the error was pinpointed, you know one of the places where a numeric check is needed in the application.

That is the end of this section, an example of using interactive reanalysis



You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers