**9. ABENDS AND DEBUGGING**

**9.1 COBOL Compiler Options**

**9.2 Assembler Options**

**9.3 Loader and Linkage Editor Options**

**9.4 COBOL Debugging Tools**

**9.5 Assembler Debugging Tools**

**9.6 Debugging Assembler Programs**

**9.7 Common Return Codes**

**9.8 VSAM and QSAM File Status Codes**

**9.9 ABEND Debugging**

**9.1 COBOL Compiler Options**

For a complete explanation of the following compiler options see the official IBM COBOL programmer's reference. There is excellent information in IBM manuals and on the internet which provide examples of the following (and other) compiler options.

The following are some of the more useful compiler options.

**SOURCE**  SOURCE is the default.

**NOSOURCE**

A program listing of the source module is to be produced.

**OFFSET** or **OFF**

**NOOFFSET** or NOOFF NOOFFSET is the default

A Condensed Listing of all the COBOL verbs in the PROCEDURE DIVISION is to be produced. The list is generated whether or not the program abends, and it will contain for each verb:

1. the statement number
2. the relative location of the verb
3. the COBOL verb

OFFSET and LIST are mutually exclusive. If both are specified, LIST is ignored and OFFSET is accepted.

**LIST**

**NOLIST** NOLIST is the default.

An assembler expansion of PROCEDURE DIVISION verbs is produced. The LIST is generated whether or not the program abends, and it will contain:

1. assembler listing of program initialization code

2. Program Global Table information

3. Constant Global Table information

4. source code assembler expansion which contains for each COBOL VERB:

source code line number and VERB

relative location in the module

assembler instruction in hex

assembler mnemonics and instruction

5. location of compiler-generated tables in the object module

6. TGT map

7. location and size of WORKING-STORAGE

8. location of literals

NOTE: OFFSET and LIST are mutually exclusive. If both are specified, the compiler rejects LIST and accepts OFFSET.

**LIB**

**NOLIB** NOLIB is the default

You must specify this as the first compiler option in the list if a copy library is referenced in the COBOL code. You must also include a SYSUT5 DD card in the compile step JCL.

**MAP**

**NOMAP** NOMAP is the default.

A glossary of all Data Division items is to be produced. It will also produce additional information on global tables, literal pools, program structure, the size of the WORKING-STORAGE section and its location within the object code. An embedded map can also be produced which lists the base locator, displacement and assembler definition in the DATA DIVISION source code. The list is generated whether or not the program abends, and it will contain for each data item:

1. source line number
2. COBOL level number
3. data name in source code
4. base locator
5. displacement from base locator
6. assembler definition
7. data type and usage
8. data definition attributes

**FLAG(x,y) or F(x,y)** FLAG(I) is the default

**NOFLAG**

Produces diagnostic messages for compiler errors with a severity level of a given level and higher. The x value controls which messages are printed at the end of the listing, and the y value controls which messages are printed line-by-line in the listing. The value specified will produce diagnostic messages for errors with that severity level or higher. The severity level of y cannot be less than that of x. In order to use FLAG(x,y), the SOURCE compiler option must also be used.

The valid options for x and y are

**I** = Informational return code = 0

**W** = Warning return code = 4

**E** = Error return code = 8

**S** = Severe return code = 12

**U** = Unrecoverable return code = 16

If we leave off the y value, no messages are printed line-by-line.

**QUOTE** QUOTE is the default.

**APOST**

This will establish the literal delimiter. When QUOTE is used literals must be enclosed in quotes, for example: "literal". To specify single quotes as the delimiter the compiler option of APOST must be specified.

**SSRANGE**

**NOSSRANGE** NOSSRANGE is the default.

SSRANGE will generate code to check effects of subscripts and indexes on tables to make sure that they are displacements which fall within the range of the table limits. The run-time option of CHECK(ON) will use this generated code to test at execution time.

**TEST(hook,symbol)**

**NOTEST** NOTEST is the default.

Used for debugging purposes. TEST will generate the line number of the statement which causes the program to abend. While this is a compiler option, it is generated at run time. In order for the statements to be flagged the "hook" option must be STMT or ALL. If you would like the possibility of obtaining a formatted dump for errors that are not trapped by Language Environment 370 specify SYM for the "symbol" option. If SYM is specified, in order to obtain a formatted dump, the run-time option of TERMTHDACT(DUMP) must also be specified.

When TEST (by itself) is used as a compiler option the default is:

TEST(ALL,SYM)

TEST may not be used when the WITH DEBUGGING MODE option is specified in a COBOL source program.

**XREF**

**NOXREF** NOXREF is the default.

**XREF(FULL)**

**XREF(SHORT)**

A sorted cross-reference listing of all data names, procedure names and program names is produced.

XREF (by itself) is XREF(FULL).

If you use XREF and SOURCE you will also get an embedded cross reference listing in the source code which gives the line number of the data item definition used in that particular COBOL statement.

XREF(SHORT) will only put items specifically reference in the

PROCEDURE DIVISION in the cross reference listing, while XREF(FULL) will include all items.

**9.2 Loader and Linkage Editor Options**

**MAP**

**NOMAP** NOMAP is the default.

Produces a MAP of the load module that lists external names and their absolute addresses on the SYSLOUT data set. The format of the module map is similar to the following:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* VS LOADER \*

\* \*

\* OPTIONS USED - PRINT,MAP,LET,CALL,RES,NOTERM,SIZE=163840 \*

\* \*

\* NAME TYPE ADDR \*

\* PROGRAM1 SD 12C010 \*

\* ILBOSR \* LR 12D61A \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NOTE: A SYSLOUT DD statement must be specified in the LOADER step or the MAP option is ignored.

**PRINT** PRINT is the default.

**NOPRINT**

This produces diagnostic messages on the SYSLOUT data set.

**LET=m** LET=4 is the default

**NOLET**

This indicates that the LOADER will try to execute a program with, at most, a severity level no greater than m. It does nor apply to the

Linkage Editor.

LET (by itself) is LET=8.

NOLET is equivalent to LET=0.

**CALL** CALL is the default.

**NCAL**

This indicates that an automatic search of the SYSLIB data set is to be made.

NOTE: If the SYSLIB DD statement is not coded in the LOADER step, the CALL option is ignored.

**RES** RES is the default.

**NORES**

Indicates that an automatic search of the Link Pack Area Queue is to be made.

NOTE: The search of the Link Pack Area is always made after processing the primary SYSLIN input, and before searching the SYSLIB data set. Thus, when the RES option is specified, the CALL option is automatically set.

**LIST** LIST is the default.

**NOLIST**

This indicates that any LINKAGE EDITOR control statements associated with the job step are to be listed.

LIST has various options.**9.3 COBOL DEBUGGING TOOLS**

**TRACING A PROCEDURE**

It is possible to trace the execution of a specific procedure or all procedures. To do this you must add PROCEDURE DIVISION code. The code that must be added is listed under the title DECLARATIVES.

PROCEDURE DIVISION.

DECLARATIVES.

DEBUG-DECLARATIVES SECTION.

USE FOR DEBUGGING ON {a procedure name or ALL PROCEDURES).

DEBUG-DECLARATIVES-PARAGRAPH.

DISPLAY MESSAGE-OUT, DEBUG-NAME, CNTR.

END DECLARATIVES.

MAIN-PROGRAM SECTION.

program source code goes here.

MESSAGE-OUT and CNTR must be defined in the WORKING-STORAGE SECTION. MESSAGE-OUT would be for the message that you want to display immediately before a paragraph is entered. In this example CNTR is also defined in the WORKING-STORAGE SECTION and can be used as a counter to see how many times a specific paragraph is entered. This variable must be incremented in the called paragraph. It is not done automatically.

If a specific procedure is being traced list that procedure name. If you want to trace all of the procedures use the ALL PROCEDURES option.

In order for this to work you must code the WITH DEBUGGING MODE option on the SOURCE COMPUTER statement in the ENVIRONMENT DIVISION. You must also code the DEBUG run-time option on the EXEC card of the GO step.

Note that this debugging option and the TEST compiler option are mutually exclusive and you may not use test on the COMPILE step.

**GENERATING A USER-INITIATED DUMP**

It is possible to called the LE/370 system and generate a formatted dump. This is done by calling the LE/370 subroutine "CEE3DMP". Note that this will produce a formatted dump, however it will not terminate execution of the program.

The format of the CALL instruction is as follows:

CALL 'CEE3DMP' USING TITLE-FLD, OPTION-FLD, FEEDBACK-FLD

The three fields referenced in the CALL statement will be defined as follows:

01 TITLE-FLD PIC X(80).

This field will contain the title to be printed at the top of each page of the formatted dump.

01 OPTION-FLD PIC X(255).

This field is used to specify the options required for this call to CEE3DMP. The default for this field as determined by LE/370 is:

TRACEBACK THREAD(CURRENT) FILES VARIABLES NOBLOCKS NOSTORAGE

STACKFRAME(ALL) PAGESIZE(60) FNAME(CEEDUMP) CONDITION ENTRY

See the LANGUAGE ENVIRONMENT/370 PROGRAMMING GUIDE for descriptions of the above options and additional option descriptions.

01 FEEDBACK-FLD PIC X(12).

This field is used for LE/370 to supply a return code regarding the success or failure of the call to CEE3DMP.

REQUIRED DD CARDS:

A CEEDUMP DD card in the Go step is required. Direct the output to the printer.

//CEEDUMP DD SYSOUT=\*

SUGGESTED COMPILER OPTIONS:

It is a good idea to include the TEST compiler option.

**EXAMPLE OF CALL TO CEE3DMP**

The following example uses the system defaults for the option field.

WORKING-STORAGE SECTION.

01 TITLE-FLD PIC X(80).

01 OPTION-FLD PIC X(255).

01 FEEDBACK-FLD PIC X(12).

PROCEDURE DIVISION.

:

:

:

100-SUBROUTINE.

MOVE 'DUMP CALL FROM 100-SUBROUTINE' TO TITLE-FLD.

CALL 'CEE3DMP' USING TITLE-FLD, OPTION-FLD, FEEDBACK-FLD.

**The DISPLAY Verb**

The DISPLAY verb is a debugging tool that makes it easier to trace the flow of data and the logic of a program. This tool enables the beginning programmer to solve logic problems. The DISPLAY verb is useful when the output is incorrect and the programmer cannot determine why. By tracing the contents of a field at critical points in the program, the programmer can determine where the problem is located.

The DISPLAY verb may be used to display literals and/or the contents of identifiers.

**Examples**

(a) Displaying a literal. The instruction:

DISPLAY 'ENTERING ROUTINE 100-READ'

will cause the following to be printed:

ENTERING ROUTINE 100-READ

(b) Displaying an identifier. The instruction:

DISPLAY NUM-1

will cause the contents of NUM-1 to be printed. If NUM-1 is a 4-digit number containing the value 100, the following would be printed:

0100

(c) Displaying a literal and an identifier. It is a good practice to label the identifiers that are to be printed so that there is no question as to which identifier is being displayed. This is done by combining literals and identifiers in the same DISPLAY statement. The following instruction illustrates this concept. If NUM-1 is a 4-digit number containing the value 100, then the instruction:

DISPLAY 'NUM-1 = ' NUM-1

will cause the following to be printed:

NUM-1 = 0100

*Note:* COMP and COMP-3 numeric identifiers will not be displayed correctly by the DISPLAY verb. To display the contents of such an identifier, move it to a DISPLAY format field of equivalent size, and then use DISPLAY to display that field.

**Debugging Tips**

Debugging programs is not an exact science. It is learned through the process of trial and error. Described below is a series of steps to take when trying to debug a program.

WHERE TO START:

1. Type in the program.

2. Submit the program to be run without printing it.

3. Check the program output for correctness (i.e., look for logic errors).

4. If instead of output, you get errors, proceed to item 5.

5. Go to the bottom of the file and then back up to the statements that tell what and where the errors are. There are two types of errors which are identified: compiler errors and execution errors. *Note:* Logic errors are not identified unless they cause execution errors. It is up to the programmer to check the output to see if the logic has achieved the desired results.

6. Remember that all corrections must be made in the original file, which must be submitted again for execution.

**COMPILER ERRORS:**

1. You should see some statements that reference a line number and the possible error in that program line.

2. Find that line in the listing to see what the problem is. (The error message from the compiler will also appear following the line on which the error occurs).

3. Things to look for:

- Spelling errors

- Syntax errors, i.e., improper use of a verb or identifier

- Using reserved words as identifiers

- Identifiers that were not defined in working storage

- Picture clauses that are too small

- Picture clauses that do not correspond with a specific verb

4. The correction must be made in the original program file. Make a note of the line number and the problem so that you can remember it.

5. SUGGESTION: Fix the errors that are obvious. Compiler errors often have a snowball effect ‑‑ one causes another. Then run the program again. Once the obvious errors are fixed, sometimes the others will be fixed in the process or, at least, will be easier to deal with when there are not so many of them.

**EXECUTION ERRORS:**

1. You will see a statement that tells you what type of execution error has been encountered. It will also tell the possible line number of the statement that caused the program to ABEND.

2. Go to that line number in the program to see what identifiers and verbs are involved. Usually, the error will be found in this line of code.

3. If that line number does not indicate the exact statement that caused the ABEND, check the lines surrounding it that also use the identifiers located on the specified line.

4. If the problem is still not clear, use the debugging tool 'DISPLAY'. Add enough DISPLAY statements to display the identifier(s) in question at various points throughout the program, and run the program again. This will track the identifier(s) and help to determine the problem.

5. Remember that corrections must be done in the original file.

6. Make sure that the return codes (RC) listed at the beginning of the JCL diagnostics for the compiler step (COB) and execution (GO) step are both '0000'. This means that the program had no errors in the compiler step and executed without an error that would cause it to ABEND. You must still check your output to see if the logic that was employed provided the correct results.

**9.4 ASSEMBLER DEBUGGING TOOLS**

**XSNAP Macro**

Note:

1. You need a //XSNAPOUT DD statement in the execution step to use these macros.

2. To continue any macro, place a non-blank character in column 72 and continue in column 16 on the next line.

XDUMP can be used under ASSIST and non-ASSIST Assemblers to obtain a dump of the general purpose registers, or a dump of a specified area of storage. The address of the XDUMP instruction will appear in the output as well as the addresses of the storage areas dumped, if any. XDUMP is a special case of XSNAP. XDUMP actually uses XSNAP to create its output.

XDUMP no operands - dumps the contents of the registers

XDUMP area,length - dumps the contents of the specified storage area for the number of bytes indicated

where area is any RX-type address and length is an absolute expression specifying a value from 1 to 4095 bytes with 4 as the default

XSNAP cannot be used under ASSIST. It can only be used under non- ASSIST assemblers. All or none of the following options may be used in any order.

XSNAP no operands - dumps the contents of the registers

**Options**

LABEL='literal' Prints an identifying label.

STORAGE=(beg-addr,end-addr[,beg-addr,end-addr...])

where beg-addr is the address of the beginning of the storage area and end-addr is the end address of the storage area. The addresses may be in any one of the following forms:

- any A-type address constant

- \*expression

where "expression" is a base displacement address or a DSECT field label (notice the asterisk)

Dumps the contents of the registers and dumps the contents of one or more storage areas.

T=NOREGS Suppresses the dumping of the registers.

**PGMDUMP Macro**

When an Assembler program abends and the DD statement for a System utility Dump has been coded, it will dump the contents of every module involved in the execution of your Assembler program including your program load module. This involves p-a-g-e-s of printed dumps. In order to debug your program you need only a dump of your program load module, the kind of dump you learned to read in CSCI 360. A dump similar to the ASSIST dump is available on our system. To access this dump of your program load module on abend, you must code the following additions in your JCL and in your source code.

1. The Assembler (ASMA90) Step SYSLIB DD statement must include a third data set:

//SYSLIB DD DSN=KC00NIU.SYS2.MACLIB,DISP=SHR

// DD DSN=SYS1.MACLIB,DISP=SHR

// DD DSN=KC00NIU.CSCI464.LOADLIB,DISP=SHR

2. The Execution Step must also contain a STEPLIB DD statement as follows:

//STEPLIB DD DSN=KC00NIU.CSCI464.LOADLIB,DISP=SHR

3. Your source code must contain the following macro statement immediately after the last line of your standard entry code:

Beginning in column 10: PGMDUMP

This statement with no operands will result in a 32-bytes per line memory dump. To obtain a 16-bytes per line memory dump, code the following:

Beginning in column 10: PGMDUMP TYPE=C

The addition of the operand 'TYPE=C' provides the shortened dump line which is easily readable on a screen. However, DO NOT PRINT the dump with the shortened lines; it uses too much paper.

To find specific storage areas in the dump, you must add the displacement of the field from the beginning of the program (found in the first column of each source code statement) to the entry point address (EPA), the address in memory where your program was loaded for execution. The ENTRY ADDRESS can be found on the Loader messages output page.

If you are using the PGMDUMP, do NOT code a SYSUDUMP DD statement.

**9.5 DEBUGGING ASSEMBLER SUBPROGRAMS**

When link editing COBOL and Assembler subprograms together, you should include provisions in your JCL for both formatted dumps and system utility dumps. The formatted dump is initiated by using the run-time option of TERMTHDACT(DUMP) which will produce a formatted dump for untrappable errors. The system utility dump must be included for the Assembler programs. In order to be able to compute the address of the abending instruction in the memory dump, you must code PARM=MAP on your link edit step. This will generate a module map of the beginning module addresses, so you will know where each program starts in the dump.

If your program abends and the formatted dump seems to indicate that it was a CALL statement for a subprogram that caused the abend, it probably means that the abend occurred within the called subprogram. If the subprogram is an Assembler module, you must calculate the address of the abending instruction:

1. The PSW AT ENTRY TO ABEND will contain the address of the instruction immediately following the instruction which caused the abend. To calculate the address of the abending instruction, subtract the value of the ILC (instruction length code) from the last three bytes of the PSW (the address of the next instruction).
2. Compute the relative address of the abending instruction by subtracting the load module entry point address from the address of the abending instruction just computed.
3. On the LINKAGE EDITOR messages page, look up the-"ORIGIN" for the external name that matches the CSECT name for the subprogram. This is the relative address of the Assembler subprogram to the entry point of the load module. Subtract this value from the relative address of the abending instruction. This is the location counter value of the abending instruction in the source listing of your Assembler program.

**9.6 COMMON RETURN CODES**

**S213**

DASD open error: Data set not found. This error occurs if you attempt to OPEN a nonexistent data set. In practice, this usually happens because the data set is not yet available or because the programmer misspelled the DDname or DSNAME specification for the data set.

**S322**

Specified or default CPU time limit exceeded. This ABEND has two common causes:

1. An endless loop which does not contain any lines of code that produce output.
2. A very long program may exceed the default CPU time limit (5 seconds at NIU). In this case, the TIME= parameter value on the job or job step should be increased.

**S722**

Estimated lines of output exceeded. This ABEND has two common causes:

1. An endless loop which contains one or more lines of code that produce output.
2. A program that produces a large amount of output can exceed the default estimated lines.

**U4093**

Has a variety of possible causes; in practice, the most common one seems to be insufficient storage. Increase the value of the REGION= parameter on the job or job step.

**3000**

This is a generic return code indicating that some sort ABEND has occurred in a COBOL program. Two major types of errors are described below.

**Program Errors**

COBOL program ABENDs will produce a message with the following format:

CEE*error-code* The system detected a *exception-type* exception. The error occurred in program unit *program-name* at entry point *entry-point* at statement *statement-number* at offset *hex-offset* at address *hex-address*.

*Exception-type* is the type of ABEND that occurred (Data, Decimal Divide, Protection, etc.). *Statement-number* is the line number of the ABENDing instruction in the COBOL program source listing. *Hex-offset* and *hex-address* can be used to find the ABENDing instruction in the assembly listing produced by the LIST compiler option.

The most common program errors that occur in COBOL are the following:

* Data exception (CEE3207S): Occurs when an operand field in an arithmetic statement does not contain a valid number.
* Decimal Divide exception (CEE3211S): Occurs when a program tries to divide by 0.
* Protection exception: Quite rare in a COBOL program; occurs when a program attempts to access an address outside the memory locations that have been assigned to it.

*Example:*

CEE3207S The system detected a Data exception. The error occurred in program unit ASSGN10 at entry point ASSGN10 at statement 115 at offset +0000050E at address 0003050E.

This message indicates that a Data exception occurred at line 115 of a COBOL program.

**File I/O Errors**

The COBOL textbook has a more complete list of file I/O errors on p. 555. Most such errors will result in a message with the following format:

IGZ0020S A logic error occurred. Neither FILE STATUS nor a declarative was specified for file *file-name* in program *program-name* at relative location *relative-location*. The status code was *status-code*.

*File-name* is the DDname of the file for which the error occurred. *Status-code* will tell you which type of error occurred. The most common status codes encountered in QSAM are listed below:

* 41: Attempt to OPEN a file that is already open.
* 42: Attempt to CLOSE a file that is not open.
* 46: Attempt to READ a nonexistent next record (in QSAM, this usually means an attempt to read past end-of-file).
* 47: Attempt to READ from a file that is not open for input.
* 48: Attempt to WRITE to a file that is not open for output.

*Example:*

IGZ0020S A logic error occurred. Neither FILE STATUS nor a declarative was specified for file INPUT in program ASSGN1 at relative location X'0474'. The status code was 46.

This message indicates that an attempt to read past end-of-file was made for the file with DDname INPUT. Unlike a program error, the line number of the READ instruction that caused the problem is not listed.

Status code 39 is a fairly common file I/O error that produces a slightly different error message:

IGZ0035S There was an unsuccessful OPEN or CLOSE of file *file-name* in program *program-name* at relative location *location*. Neither FILE STATUS nor an ERROR declarative were specified. The status code was *status-code*.

Status code 39 occurs when there is a conflict between the fixed attributes of a file (such as LRECL, BLKSIZE, RECFM, etc.) and what the programmer has coded as for the FD or JCL for that file. The most common cause is coding an incorrect record length in the FD for an input file.

**9.7 VSAM and QSAM File Status Codes**

These are taken from various sources; some are IBM standards and some are ANSI standards. They do not include status codes for operations involving optional files or files closed with locks. Many of these status codes have the same or similar meanings for QSAM.

|  |  |
| --- | --- |
| Value | Meaning |
| 00  02  04 | Successful Completion |
| Successful completion (of any I/O operation); or in alternate key processing with a nonunique alternate key, the READ of the last matching record.  In alternate key processing with a nonunique alternate key: a successful READ in alternate key processing with more matching records yet to be read; or a successful WRITE or REWRITE creating a duplicate value for an alternate key (for which duplicates are allowed).  Successful READ, but the logical record's size is too large or too small to match the specified size. |
| 10  14 | Unsuccessful Completion because of At End |
| End-of-file detected on READ in sequential processing.  Sequential READ failure for RRDS: RRN has more significant digits than specified for the relative key. |
| 21  22  23  24 | Unsuccessful Completion because of Invalid Key |
| Sequence error for a sequentially accessed indexed file: key value changed between READ and REWRITE, or out of order.  WRITE failure: slot in use (RRDS) or attempt to create a record with a duplicate primary (or unique alternate) key (KSDS).  An attempt was made to randomly access a record that does not exist in the file, or a START or random READ statement was attempted on an optional input file that was not present.  WRITE failure: an attempt to write beyond the physical end of an ESDS or RRDS cluster; or a sequential WRITE failure for RRDS (RRN has more significant digits than specified for the relative key). |
| 30  34  35  37  39 | Unsuccessful Completion because of Permanent I/O Error |
| Permanent I/O error with no further information available.  Permanent error due to boundary violation: attempt to WRITE beyond the physical end of an ESDS cluster.  Permanent error because of an attempt to OPEN (for INPUT, I-O, or EXTEND) a file not present.  Permanent error because of an attempt to OPEN a file for a mode not supported by that file.  Permanent error due to OPEN failure; conflict in cluster setup data such as organization, record key, etc. |

|  |  |
| --- | --- |
| 41  42  43  44  46  47  48  49 | Unsuccessful Completion because of logic error |
| OPEN failure: file already open.  CLOSE failure: file not open.  REWRITE or DELETE failure in sequential processing: not preceded by a successful READ.  Boundary violation; attempt to REWRITE a record not the same size as the record being replaced, or to WRITE or REWRITE a record whose size does not match the limits defined for the file (as in a variable-length record).  Sequential READ failure because the current file pointer is undefined: preceding READ or START failed or a preceding READ hit EOF.  READ or START failure: file not open for INPUT or I-O  WRITE failure: file not opened for OUTPUT, I-O, or EXTEND  DELETE or REWRITE failure: file not opened for I-O. |
| 90  91  92  95  96 | Miscellaneous Errors |
| Undocumented error with no further information available.  Password failed (VSAM).  Logic error.  File information wrong or incomplete (VSAM only).  No DD in the JCL for this file (VSAM only). |

**9.9 ABEND Debugging**

Programs can sometimes terminate abnormally, either by program interruption caused by a runtime exception or by a user-designated interruption using the IBM ABEND macro. In both cases, a system memory dump, or sys-dump, consisting of a listing of the job’s JCL, the program source listing itself, and the contents of the registers and the program’s memory storage is produced along with lots of other information, some of which is useful and a lot which is not.

In the case of an interruption forced by a user-designated use of the IBM ABEND macro, the instruction or place in the program source listing the instruction which cause ABEND by examining the user interruption code. Generally, the first page of the dump will display the information about registers contents and PSW contents. There are two ways to find out the ABENDing instruction using the following dump:

1 J E S 2 J O B L O G -- S Y S T E M S 0 W 1 -- N O D E Z O S K C T R

0

20.59.30 JOB03170 ---- WEDNESDAY, 04 NOV 2015 ----

20.59.30 JOB03170 IRR010I USERID KC03Q17 IS ASSIGNED TO THIS JOB.

20.59.30 JOB03170 IEF677I WARNING MESSAGE(S) FOR JOB KC03Q17A ISSUED

20.59.30 JOB03170 ICH70001I KC03Q17 LAST ACCESS AT 20:57:05 ON WEDNESDAY, NOVEMBER 4, 2015

20.59.30 JOB03170 $HASP373 KC03Q17A STARTED - INIT 2 - CLASS A - SYS S0W1

20.59.31 JOB03170 IEA995I SYMPTOM DUMP OUTPUT 212

212 SYSTEM COMPLETION CODE=0C7 REASON CODE=00000000

212 TIME=20.59.30 SEQ=06132 CPU=0000 ASID=0033

212 PSW AT TIME OF ERROR 078D0000 00015028 ILC 6 INTC 07

212 ACTIVE LOAD MODULE ADDRESS=00015000 OFFSET=00000028

212 NAME=\*\*GO

212 DATA AT PSW 00015022 - FA22C086 C08958DD 000498EC

212 GR 0: 00000000 1: 000101B0

212 2: 000101B0 3: 1ED01037

212 4: 1ED02036 5: 1ED001E4

212 6: 00000000 7: 000150B0

212 8: 0000004E 9: 1ED00318

212 A: 03A8603E B: 03A8503F

212 C: 40015022 D: 00015060

212 E: 80FDA490 F: 00015000

212 END OF SYMPTOM DUMP

20.59.31 JOB03170 IEF450I KC03Q17A JSTEP02 - ABEND=S0C7 U0000 REASON=00000000

20.59.31 JOB03170 $HASP395 KC03Q17A ENDED

1. **Using the content of the PSW**

We can find out the instruction which caused the ABEND by observing the content of PSW at the time of the error. It is very important to note that the content on PSW shown in this dump is in simulated format and we might have to interpret it differently.

212 PSW AT TIME OF ERROR 078D0000 00**015028** **ILC 6** **INTC 07**

* The interruption code in PSW is not shown in lower 2 bytes of first full word of PSW.
* The interruption code is shown separately as **INTC 07** i.e. Data Exception.
* The instruction length code is also not shown in the contents of PSW.
* The instruction length code is shown under ILC is in **bytes**. In this example, **ILC 6** which means the previously executed instruction length is 6 bytes.
* The address of next instruction, provided in the PSW, is of size 24-bit and it can be found at last 3 bytes of the content of PSW – **015028.**
* Since the actual system does not use exact 24-bit addressing scheme, the address represented in PSW is simulated or virtual address. We have to convert this address to the logical address which we can use to locate the instruction in the source code listing.
* To do so first locate the assigned or virtual 24-bit address of the active module (address of the start of the program). In this case it is 015000.

212 ACTIVE LOAD MODULE ADDRESS=**00015000** OFFSET=00000028

* The subtraction of this address from the address of next instruction will give us the logical address.

015028 – 015000 = 28 – The address of the next instruction to be executed.

[Note: This is hexadecimal computation]

* Now, if we subtract the previous instruction length from this address i.e. ILC (6 bytes), we will get the logical address of the instruction which ABEND the program.

28 – 6 = 22 – The logical address of ABEND instruction.

[Note: This is hexadecimal computation]

* We can refer to this location in the source code listing provided in the system dump to figure out the actual instruction. The source code listing is given below.

1 Page 3

Active Usings: None

0 Loc Object Code Addr1 Addr2 Stmt Source Statement HLASM R6.0 2015/11/04 20.59

0 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ASMDUMP CSECT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2 \* \*

3 \* PROGRAM - ASMDUMP \*

4 \* USE - DEMO PROGRAM FOR ANALYSIS OF DUMP \*

5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

000000 00000 000AE 6 ASMDUMP CSECT BEGIN ASMDUMP=

7 \*

8 XSAVE BR=12,SA=MAINSAVE,TR=NO STANDARD ENTRY LINKAGE

-000000 11+ DS 0H DEFINE LABEL, MAKE SURE ALIGNED

R:F 00000 12+ USING \*,15 FOR TEMPORARY ADDRESSIBILITY

000000 47F0 F00E 0000E 14+ B 14(,15) BRANCH AROUND ID

000004 09C1E2D4C4E4D4D7 15+ DC AL1(9),CL9'ASMDUMP'

00000E 90EC D00C 0000C 16+ STM 14,12,12(13) SAVE STANDARD REGISTER SET

000012 50D0 F064 00064 17+ ST 13,MAINSAVE+4 SAVE OLD POINTER IN NEW AREA

000016 18CD 18+ LR 12,13 MOVE OLD POINTER OVER

000018 41D0 F060 00060 19+ LA 13,MAINSAVE ADDRESS OF NEW SAVE AREA

00001C 50DC 0008 00008 20+ ST 13,8(12) SAVE NEW POINTER IN OLD AREA

000020 05C0 21+ BALR 12,0 SET UP NEW BASE REGISTER

23+ DROP 15 CLEAN UP USING SITUATION

R:C 00022 24+ USING \*,12

0 26 \*

**000022 FA22 C086 C089 000A8 000AB 27 AP NUM1(3),NUM2(3)**

28 \*

29 XRETURN RC=0,TR=NO STANDARD EXIT LINKAGE

0000028 58DD 0004 00004 31+ L 13,4(13) RESTORE PREVIOUS SAVE AREA POINT

00002C 98EC D00C 0000C 32+ LM 14,12,12(13) STANDARD REGISTER RESTORATION

000030 41F0 0000 00000 33+ LA 15,0 PUT RETURN CODE IN 15

000034 07FE 34+ BR 14 RETURN NORMALLY TO CALLER

0 36 \*

000038 37 LTORG LTORG TO ORGANIZE LITERALS

38 \*

39 \*\*\*STORAGE ARE FOR MAIN\*\*\*

40 \*

000038 00038 00040 41 ORG ASMDUMP+((\*-ASMDUMP+31)/32)\*32

000040 C8C5D9C540C9E240 42 DC C'HERE IS THE STORAGE FOR THIS PGM'

43 \*

000060 FFFFFFFFFFFFFFFF 44 MAINSAVE DC 18F'-1' MAINSAVE FOR STANDARD LINKAGE

0000A8 00000C 45 NUM1 DC PL3'0'

0000AB C1C2C3 46 NUM2 DC CL3'ABC'

47 \*

000000 48 END ASMDUMP

* The first column in the source code listing - Loc - is the location counter in hex which can be considered as the logical address for the instructions in program.
* We can refer to location 22 to get following instruction

000022 FA22 C086 C089 000A8 000AB 27 AP NUM1(3),NUM2(3)

* This is instruction terminated the program abnormally due to data exception.

**Few Notes on Source Code Listings**

* Immediately after the location counter, the encoded format of the each instruction is shown with opcode and both operand address. For example:

FA22 C086 C089 – 6 byte long.

* The instructions which are started with + sign after statement number or count (under Stmt column) in this listing is the expanded macro code which is automatically generated by the macro.

XSAVE BR=12,SA=MAINSAVE,TR=NO STANDARD ENTRY LINKAGE

* The location counter may change because of the ORG instruction.

1. **Using data at PSW (Data of the location pointed by the PSW)**

We can also find out the ABEND instruction by data at PSW. This information provides the **memory content** of the **previous instruction**. In other words, this is encoded representation of the instruction which caused the ABEND.

212 DATA AT PSW 00015022 - **FA22C086 C089**58DD 000498EC

* Note the virtual address specified 015022 (Logical address 22) and the content which is

FA22C086 C089

(Consider only first 6 byte because ILC is 6)

* We can decode this information to identify the instruction. The first byte i.e. FA is the opcode of the instruction. Using IBM reference summary, we can find out that this is opcode of the AP instruction.
* Since, the AP instruction is of SS format with two different length, we can decode the 6-byte instruction as,

|  |  |  |  |
| --- | --- | --- | --- |
| Encoded Content  (All data is in Hexadecimal) | Size | Meaning | Decoded Content  (Numbers are in Decimal) |
| FA | One byte | Opcode | AP |
| 2 | Half byte | Length of first operand | 3 [Add 1 because it is length] |
| 2 | Half byte | Length of the second operand | 3 |
| C | Half byte | Base register for first operand | 12 |
| 086 | One & half byte | Displacement (in Hex) of first operand | 134 |
| C | Half byte | Base register for second operand | 12 |
| 089 | One & half byte | Displacement of second operand | 137 |

[Refer the MACHINE INSTRUCTION FORMATS in IBM Summary to find out more details about instruction formats].

Decoded instruction – AP 134(3,12),137(3,12)

* We can use the hexadecimal displacement of both operands to locate them in source listing.

Just consider 086 as the logical address of the operand 1 and 089 as the logical address of the operand 2.

* If you see the source listing, you can find that the location provided for operand 1 and operand 2 are different from what we have decoded. This behavior is caused by the macro XSAVE. The reason for the difference is that XSAVE sets up the base register, R12, to point not at the CSECT but X'22' (34) bytes beyond it.
* To calculate the actual address of operand in the source listing we have to add 22 hex in current value of hex operand.

Location of Operand 1 – 086 + 22 = 0A8

Location of Operand 2 – 089 + 22 = 0AB

**Memory DUMP**

Once you get the details of the ABEND instruction, you can use the instruction code to fix the error. In this case the instruction code is 07 i.e. Data Exception for instruction AP. The reason behind this exception is the data from either one or both operands is not in valid packed decimal format.

It is simple observation that the second operand – NUM2 – in above program holds invalid data.

0000AB C1C2C3 45 NUM2 DC CL3'ABC'

Since C1C2C3 is not a valid 3 byte packed decimal number, the error is occurred. We can easily spot this error because the given program is short one, and the data at NUM2 is not changed prior to the AP instruction. But in many cases we might have to look the content of memory location pointed by the any operand (In this case the three bytes data at location 0A8 and 0AB) at the time of ABEND. We can see this memory content or memory dump from the system dump.

Somewhere down to the source listing we can find out the memory dump at the time of the ABEND as follows,

1JOB KC03Q17A STEP JSTEP02 TIME 205930 DATE 15308 ID = 000 PAGE 00000020

REGISTERS AT ENTRY TO ABEND

FLOATING POINT REGISTER VALUES

FPC 00000000

0-3 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

4-7 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

8-11 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

12-15 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

GPR VALUES

0-3 00000000 000101B0 000101B0 1ED01037

4-7 1ED02036 1ED001E4 00000000 000150B0

8-11 0000004E 1ED00318 03A8603E 03A8503F

12-15 40015022 00015060 80FDA490 00015000

ACCESS REGISTER VALUES

0-3 00000000 00000000 00000000 00000000

4-7 00000000 00000000 00000000 00000000

8-11 00000000 00000000 00000000 00000000

12-15 00000000 00000000 00000000 00000000

64-BIT GPR VALUES

0-3 00000000 00000000 00000000 000101B0 00000000 000101B0 00000000 1ED01037

4-7 00000000 1ED02036 00000000 1ED001E4 00000000 00000000 00000000 000150B0

8-11 00000000 0000004E 00000000 1ED00318 00000000 03A8603E 00000000 03A8503F

12-15 00000000 40015022 00000000 00015060 00000000 80FDA490 00000000 00015000

0000F1A0 40404040 40404040 40404040 40404040 \* \*

0000F1C0 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 \* \*

LINES 0000F1E0-0000FA40 SAME AS ABOVE

0000FA60 40404040 40404040 40404040 40404040 F1404040 40404040 4040405C 5C5C4040 \* 1 \*\*\* \*

0000FA80 C4C1E3C1 40E2C5E3 40E2E4D4 D4C1D9E8 40405C5C 5C404040 40404040 40404040 \*DATA SET SUMMARY \*\*\* \*

0000FAA0 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 \* \*

LINES 0000FAC0-0000FB40 SAME AS ABOVE

0000FB60 40404040 40C4C4D5 C1D4C540 404040C3 D6D5C3C1 E3404040 C6C9D3C5 40C9C4C5 \* DDNAME CONCAT FILE IDE\*

0000FB80 D5E3C9C6 C9C3C1E3 C9D6D540 40404040 40404040 40404040 40404040 40404040 \*NTIFICATION \*

0000FBA0 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 \* \*

LINES 0000FBC0-0000FCA0 SAME AS ABOVE

0000FCC0 40404040 40404040 40404040 40404040 40400000 00000000 00000000 00000000 \* ..............\*

0000FCE0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 0000FD00-0000FEA0 SAME AS ABOVE

0000FEC0 C9C5E6C2 D9C9D640 00000174 01000000 D6D7C5D5 C4C9C1C7 00000000 00000000 \*IEWBRIO ........OPENDIAG........\*

0000FEE0 C9C5E6C4 C9C1C740 00000000 00000000 00000000 00000000 00000000 00000000 \*IEWDIAG ........................\*

0000FF00 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 0000FF20-0000FF40 SAME AS ABOVE

0000FF60 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000FFDC \*................................\*

0000FF80 00000000 00000000 00000000 00000000 00000001 00004000 00000001 84000001 \*...................... .....d...\*

0000FFA0 00000000 C9C5E6C4 C9C1C740 02000050 00000001 00000001 00000000 00000000 \*....IEWDIAG ...&................\*

0000FFC0 00000001 00000001 00000001 00000000 00000001 00000000 00000001 C4C3C2C5 \*............................DCBE\*

0000FFE0 00380000 00000000 00000000 00000000 10000000 00000000 00000000 00000000 \*................................\*

00010000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 1ED24638 \*.............................K..\*

00010020 83C1004A 1ED24918 00000000 83B7B816 83B7A2E0 00000000 C9C5E6C2 D9C9D640 \*cA.ۮK......c...c.s\....IEWBRIO \*

00010040 00000174 01000000 D6D7C5D5 C4E4D4D7 00000000 00000000 40404040 40404040 \*........OPENDUMP........ \*

00010060 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00010080-00010160 SAME AS ABOVE

1JOB KC03Q17A STEP JSTEP02 TIME 205930 DATE 15308 ID = 000 PAGE 00000021

00010180 00000000 00000000 00000000 00000000 00000000 1ED24638 83C1004A 1ED248F0 \*.....................K..cA.ۮK.0\*

000101A0 00000000 00000000 00000000 00000000 800101B4 00000000 00000238 01000000 \*................................\*

000101C0 54000002 E2E8E2D3 C9D54040 00000000 1ED24638 00500050 00010001 0050002C \*....SYSLIN .....K...&.&.....&..\*

000101E0 00000000 00000000 00000000 00014E9C 00000000 00006948 00014F12 00000000 \*..............+...........|.....\*

00010200 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00010220-000102E0 SAME AS ABOVE

00010300 0024C1D9 00000000 00000000 00000000 00014E98 000000E0 00010001 00000000 \*..AR..............+q...\........\*

00010320 00000000 00010384 00000000 05310000 05000000 00000000 00006940 00004000 \*.......d................... .. .\*

00010340 00000001 84CA10A0 000103BC E2E8E2D3 C9D54040 0A004800 00000001 0ACA10A6 \*....d.......SYSLIN ...........w\*

00010360 00000000 00000000 00000001 00000001 00000001 00000000 00000001 00000000 \*................................\*

00010380 00000001 C4C3C2C5 00380000 00000000 00000000 00000000 10000000 00000000 \*....DCBE........................\*

000103A0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 05CA10AC \*................................\*

000103C0 11CA10B6 160103D0 93010300 00000000 80000000 00014E9C 83C065FC 83C063E2 \*.......}l.............+.c{..c{.S\*

000103E0 83C05E5E 83C06282 00000000 00000000 00000000 00000000 00000000 00000000 \*c{;;c{.b........................\*

00010400 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00010420-00010FE0 SAME AS ABOVE

00011000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00011020-00011180 SAME AS ABOVE

000111A0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

00014000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00014020-00014E60 SAME AS ABOVE

00014E80 00000000 00000000 00000000 00000000 00000000 00000000 00E08017 E2E8E2F1 \*.........................\..SYS1\*

00014EA0 F5F3F0F8 4BE3F2F0 F5F9F3F0 4BD9C1F0 F0F04BD2 C3F0F3D8 F1F7C14B D6C2D1D4 \*5308.T205930.RA000.KC03Q17A.OBJM\*

00014EC0 D6C44BC8 F0F14040 40404040 40404040 00001910 00000000 00000000 00000200 \*OD.H01 ................\*

00014EE0 00000000 00000000 00000000 73013400 00000041 00000000 00000000 00000000 \*................................\*

00014F00 00000000 00000000 00000000 00000000 0001E3C5 D4D7F0F0 40404040 40404040 \*..................TEMP00 \*

00014F20 40404040 40404040 40404040 40404040 00000000 00000000 00000000 00000008 \* ................\*

00014F40 00000000 00000000 00000100 00000000 00280000 00000000 00000000 00000000 \*................................\*

00014F60 00000000 00007FF8 00000000 00000000 00000000 00000000 00014F80 00010080 \*......"8..................|.....\*

00014F80 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

LINES 00014FA0-00014FE0 SAME AS ABOVE

00015000 47F0F00E 09C1E2D4 C4E4D4D7 404090EC D00C50D0 F06418CD 41D0F060 50DC0008 \*.00..ASMDUMP ..}.&}0....}0-&...\*

00015020 05C0FA22 C086C089 58DD0004 98ECD00C 41F00000 07FE0000 00000000 00000000 \*.{..{f{i....q.}..0..............\*

00015040 C8C5D9C5 40C9E240 E3C8C540 E2E3D6D9 C1C7C540 C6D6D940 E3C8C9E2 40D7C7D4 \*HERE IS THE STORAGE FOR THIS PGM\*

00015060 FFFFFFFF 00006A40 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF \*....... ........................\*

00015080 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF \*................................\*

000150A0 FFFFFFFF FFFFFFFF 00000CC1 C2C30000 00015000 5C5CC7D6 40404040 00000000 \*...........ABC....&.\*\*GO ....\*

000150C0 00000010 00000001 00015000 000000E0 00000000 00000000 00000000 00000000 \*..........&....\................\*

000150E0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \*................................\*

This is very large memory dump and it can be very difficult to find out exact location of any memory location. We can use ORG trick described in the document, *‘Assembler Programming Tips & Debugging’*, posted on blackboard to make it easy. We can search the message, *‘HERE IS THE STORAGE FOR THIS PGM’*, from the right most part of the memory dump because the storage area in main is started with this message declaration (Note : The right most part in the memory dump will display the printable content of the memory locations).

**00015040** C8C5D9C5 40C9E240 E3C8C540 E2E3D6D9 C1C7C540 C6D6D940 E3C8C9E2 40D7C7D4 \***HERE IS THE STORAGE FOR THIS PGM**\*

00015060 FFFFFFFF 00006A40 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF \*....... ........................\*

00015080 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF \*................................\*

**000150A0** FFFFFFFF FFFFFFFF 00000CC1 C2C30000 00015000 5C5CC7D6 40404040 00000000 \*...........ABC....&.\*\*GO ....\*

The first column in this dump is the virtual address of the memory location. Again we can convert this address to logical address by subtracting the starting virtual address of the program or address of the active load module (by subtracting 015000). We can observe that the address of main storage is started from 40(hex). We can follow the next few lines to locate address starting from A0.

Address Displacements (in Hex)

0-3 4-7 8-B C-F 10-13 14-17 18-1B 1C-1F

000150A0 FFFFFFFF FFFFFFFF 00000CC1 C2C30000 00015000 5C5CC7D6 40404040 00000000

In this memory dump each group of 8 hexadecimal digits represents the half-word (4 byte) content of memory locations starting from the address specified at first column i.e. A0. So the first group shows the byte (2 hex digits) contents of memory location A0 to A3, the next group shows the memory content of A4 to A7 and so on. Each line will shows total of 32 bytes (32 hex digit pair of each byte) starting with address specified in first column to the address with hex displacement of 1F from that address.

The above line starts with address is A0. So memory location of last byte (last 2 hex digits of the line) is,

A0 + 1F = BF.

Hence next line will start with address C0 followed by E0 and so on.

In this program to observe the content of the both operands we will need the location of both operands and length of each operand. We can find the content of operand as follows,

|  |  |  |  |
| --- | --- | --- | --- |
| Operand | Location | Length (in bytes) | Operand Data |
| 1 – NUM1 | A8 | 3 | 00000C |
| 2 – NUM2 | AB | 3 | C1C2C3 |

In this way we can analyze the system dump output of the program to find out the cause of the error.

This document was written by Ashish Kharde, a graduate student enrolled in CSCI 565 at NIU in fall 2015. It was produced under guidance of

*Geoffrey D. Decker*

*Instructor*

*Department of Computer Science,*

*Northern Illinois University*

References

* *Lecture Notes, Manual & Slides from course CSCI-565 by Prof. Geoffrey D. Decker*
* *IBM z/Architecture Reference Summary*
* *Assembler Language with ASSIST and ASSIST/I, by Overbeek, Ross A., and W. E. Singletary. Publisher - Prentice-Hall, Inc.*