

## Dump Solving

Introduction .....	1
Entry Points .....	2
ABENDS .....	3
Job Output following an ABEND .....	4
The Job Log .....	5
Abend Codes .....	6
Other Important Fields .....	7
Solving the Dump .....	8
Check ABEND code .....	8
Calculate the Relative address of the abending instruction .....	9
Finding the line of COBOL code .....	10
Finding the Contents of Fields .....	13
Base Locators .....	14
The VS COBOL II Abend Information File .....	15
Calculating the Address of the Field .....	16
Interrogating the Dump .....	17
Appendix A .....	19

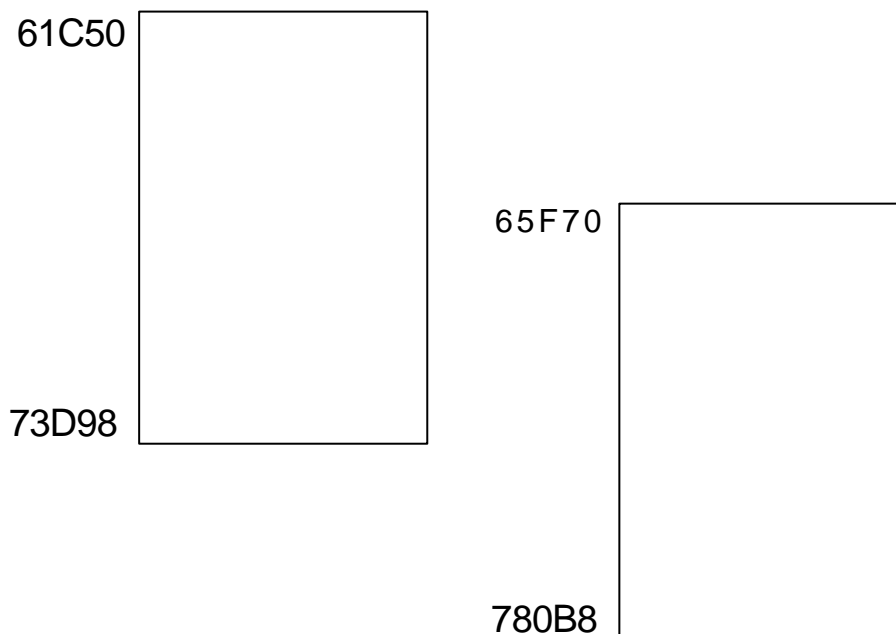


*Dump Solving*

## Introduction

When a program is to be executed the operating system brings the program into main storage. Programs running under the **MVS** operating system are said to be **relocatable**; that is, each time a program is brought into storage to be executed it could be placed in a different area of memory, depending on what was available at the time.

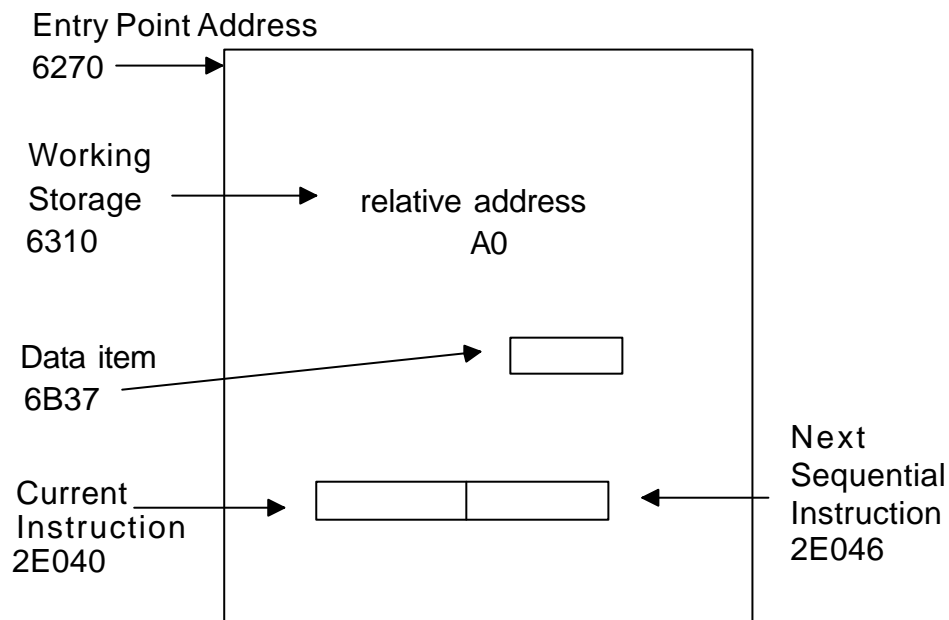
Each area, or byte, of main storage has an **address** associated with it, so that the operating system can locate information quickly and easily. When a program is fetched into memory for execution it is placed at one of these addresses. It is this address that may change each time the program is run. Note that all addresses are in hexadecimal notation. For example,



## Entry Points

The address where the first byte of the program is located, in the previous examples 61C50 and 65F70, is known as the **Entry Point Address**, the **Base Address** or the **Origin** of the program.

The program in main storage does not only consists of the program code; storage space is set aside for the working storage data areas which you define, the file descriptions, and various internal registers and tables which are set up by the operating system to contain relevant information required at execution. Each of these constituents has its own **absolute** address within store, and will also have an address **relative** to the programs base address. This is known as the **displacement** from the base address. For example,



## Abends

When a program is being executed many things can go wrong with it; invalid input data which has not been accounted for in the program, logic errors, incorrect allocation of output files, incorrect reference to a subroutine such as BPS routines and so on. When an error occurs in the execution of a program it is often severe enough to prevent the program from continuing execution and it **AB**normally **ENDS**, or **ABENDS**. Such a premature termination of processing usually causes the production of a **DUMP**.

A dump is designed to help the programmer pinpoint the error and give an indication as to the probable cause and suggested remedies. It does this by producing a copy of the contents of main storage at the time of the abend (in Hex. format), together with the contents of internal registers and control blocks - which keep track of what is happening in the system.

## Job Output Following an ABEND

As with a job which has run without incident there are several files following an abend, some of which are similar to those of a working program.

- File 1:** This contains the job log. It is in a similar format to the job log of a working program but also includes some additional important information about the abend.
- File 2:** Contains the JCL listing.
- File 3:** Contains the device allocation messages. It contains condition codes for each job step and will contain a duplication of the abend information found in the job log.
- File 4:** Contains the source code listing for the program together with the output from the compilation step of the job (as covered in the compilation tutorial). Much of this information is useful in solving the dump.
- File 5:** The Linkage editor report (from the second step of the job).
- File 6:** The largest file, containing the body of the dump itself. Abend statistics, internal tables, registers and memory contents. You should never print off the contents of this file!
- File 7:** Any report file(s) produced or partially produced prior to the abend.
- Last File:** Contains additional Cobol II abend information, which may be required to solve the problem. This may appear before or after any output reports produced.

## The Job Log

In addition to the usual information, is the program abends further information and statistics can be found here. For example,

```

>
> JOB(DEVSRTTR1,5485) SCRL CSR COLS 00012 00084 F 01 P 0001
...+...2...+...3...+...4...+...5...+...6...+...7...+...8...
===== T O P =====
000001 J E S 2 J O B L O G -- S Y S T E M A A A A -- N O D E
000002 B05485 IEF097I DEVSRTTR1 - USER DEVSRTTR ASSIGNED
000003 B05485 IEF677I WARNING MESSAGE(S) FOR JOB DEVSRTTR1 ISSUED
000004 B05485 CHASP373 DEVSRTTR1 STARTED - INIT TJ - CLASS T - SYS AAAA
000005 B05485 IEA995I SYMPTOM DUMP OUTPUT
000006 A SYSTEM COMPLETION CODE=0C7 REASON CODE=00000007
000007 TIME=11.27.49 SEQ=50404 CPU=0000 ASID=0055
000008 B PSW AT TIME OF ERROR 078D1000 83001422 C ILC 6 D INTC 07
000009 E ACTIVE LOAD MODULE=PROG1 F ADDRESS=03000DA8
OFFSET=0000067A
000010 H DATA AT PSW 0300141C - F9218010 C01E47D0 B270F922
000011 I GPR 0-3 83001E06 0002AE70 03020228 03020340
000012 GPR 4-7 03020228 83001338 000052EC FD000000
000013 GPR 8-11 0002AE70 03020560 03000E50 030011C0
000014 GPR 12-15 03000E28 03020028 7000AD14 8000AD24
000015 END OF SYMPTOM DUMP
000016 B05485 J +IGZ057I An ABEND was intercepted by the COBOL execution
time 000017 B05485 + handler. It is described by a corresponding
IEA995I

```

- A - Abend Code
- B - Program Status Word at time of error
- C - Instruction Length Code
- D - INTerrupt Code
- E - Active Load Module
- F - Entry Point Address
- G - Offset of Interrupt Relative to the Entry Point
- H - Data at Program Status Word
- I - General Purpose Register
- J - Runtime Message

## Abend Codes

These are also known as **completion codes**. There are two mutually exclusive types of abend code:

**System Code** The system itself has terminated the execution of the program after recognising a fault. The code indicates the type of error, and is often accompanied by a message which is designed to help determine the problem. Several of these codes are quite common and will be responsible for the majority of abends and as such probable causes can be quickly deduced, however some of the less frequently encountered codes can be found in relevant system messages manuals.

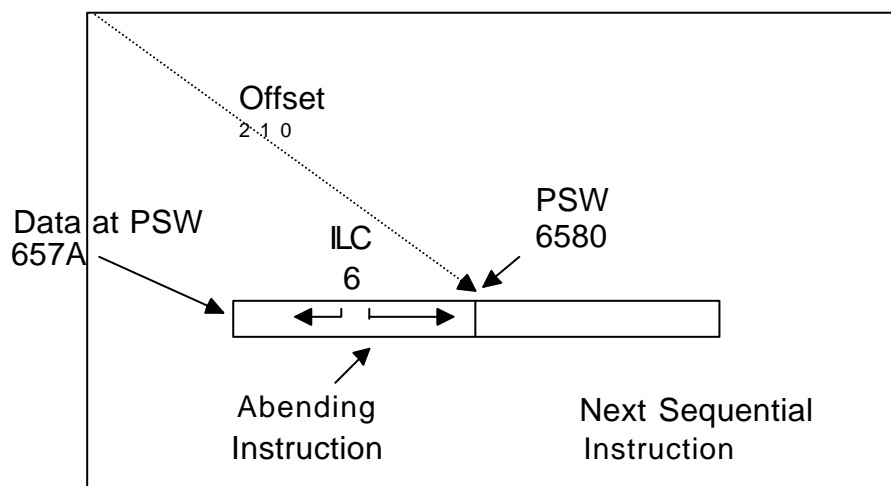
**User Codes** These occur when the user program recognises an error and issues an abend itself. These errors are catered for within the program code and the program usually outputs an explanatory message to the console before terminating processing itself. Examples of such errors would be invalid return codes from a subroutine, invoking of MAGABEND routine and incorrect use of a BPS routine. As you have seen BPS has its own error codes which it issues when such an abend occurs.

Some of the more common abend codes and a brief description of the possible reasons are listed in APPENDIX A. More information about abend codes and messages can be found in 'VS COBOL II Application Programming: Debugging' and 'OS/VS System Messages' manuals.

## Other Important Fields

- Program Status Word** The PSW at time of Error field an 8 byte hex. address. The last 4 bytes of this (ignoring the first half byte) is the hex. address of the next sequential instruction to be executed.
- I.L.C** The Instruction Length Code gives the length of the currently executing instruction (the abending instruction) in bytes.
- Address** This field denotes the entry point or origin of the program, i.e. it's start address within memory.
- Offset** The offset field denotes the address of the **next** sequential instruction to be executed. This hex. address is relative to the origin of the program.
- Data at PSW** The first 4 bytes of this field give the absolute address of the abending instruction. The other bytes give the actual instruction in hex.

Entry Point  
6370



For example,



## Solving the Dump

### ***Check ABEND code***

The first step in solving a dump is to look in file 1 (the Job Log) and check the abend code. There are many different types of abend and with experience you will learn to recognise possible causes of each. The most common of these is 0C7 or **data exception error**. With practice you can usually solve these types of abend quite quickly. This tutorial is designed to take you through the first steps of solving an 0C7 dump - this information will not necessarily apply to other types of dump.

### **Calculate the Relative address of the abending instruction**

We now need to establish which line the COBOL program caused the abend. This involves finding the relative address of the abending instruction.

The easiest method of doing this is to use the **OFFSET** and **ILC** fields. The **OFFSET** denotes the relative address of the next sequential instruction to be executed. In order to calculate the relative address of the abending instruction we can subtract the **ILC** (length in bytes of the abending instruction) from the **OFFSET** giving us the address of the abending instruction relative to the origin of the program.

#### **Relative Address of abending = offset - ILC instruction**

For example,

$$67A - 6 = 674$$

Thus the abending instruction can be found at hex. 674 from the start of the program in memory.

You can also calculate this address by subtracting the start address of the program (in the **ADDRESS** field) from the address of the abending instruction (**DATA AT PSW** field) giving the relative address of the abending instruction. For example,

$$290141C - 2900DA8 = 674$$

There are other methods but we will not cover these in this tutorial.

### ***Finding the line of COBOL code***

Once you have obtained the relative hex. address of the abending instruction you have to find out which statement within your COBOL program code can be found at this address.

This can be achieved by using the CONDENSED VERBS LISTING which can be found in File 4 of your job. This table does not have a title so may be difficult to locate at first, though with practice you will now know where to look immediately. It can be found directly following the Literal Pool Map which is located 3 or 4 pages from the end of your source code listing in File 4. There are three columns of information spread across the page. Each column is further subdivided into LINE, HEXLOC and VERB. For example,

LINE #	HEXLOC	VERB	LINE #	HEXLOC	VERB
000129	0004FE	PERFORM	000141	0004FE	OPEN
000145	000524	WRITE	000146	000550	MOVE
000149	000578	PERFORM	000227	000578	READ
000151	0005BC	IF	000152	0005C6	MOVE
000154	0005D6	WRITE	000131	000602	PERFORM
000162	000610	MOVE	000163	00061A	MOVE
000166	000620	PERFORM	000186	000626	IF
000188	000654	MOVE	000190	000664	MOVE
000194	000688	MOVE	000168	000694	MOVE
000171	0006C8	PERFORM	000201	0006C8	MULTIPLY
000203	0006EA	IF	000204	0006F2	MOVE
000209	000706	MOVE	000211	00070C	SUBTRACT
000215	00073E	SUBTRACT	000173	00075C	MOVE
000175	00078C	MOVE	000177	0007A4	MOVE
000180	0007D4	PERFORM	000227	0007D4	READ
000133	000822	PERFORM	000221	000822	CLOSE

**LINE** refers to a line within the COBOL source code compiler listing in file 4.

**HEXLOC** is the relative hex. address of the COBOL instruction from the origin of the program in memory

**VERB** gives the COBOL verb that is coded on that line.

Look at the values in the HEXLOC columns. The columns are read left to right across the page, and top to bottom. Find the hex. value which is closest to but less than the relative address of the abending instruction calculated previously. (There may be a HEXLOC value which is equal to the address but this is not always the case.) The corresponding VERB and LINE entries can be used to find the abending line of source code in the compiler listing. For example, for a hex. relative address of 674.

LINE #	HEXLOC	VERB
000144	000524	MOVE
000147	000550	WRITE
000229	0005B6	MOVE
000153	0005D6	MOVEPage13
000161	000606	MOVE
000164	00061A	MOVE
000187	000640	IF
000193	000674	IF
000169	0006B0	MOVE
000202	0006E0	IF
000206	0006FC	MOVE
000213	000722	MULTIPLY
000174	000774	MOVE
000178	0007A4	WRITE
000229	000812	MOVE
000135	000848	GOBACK

The associated line of code would be

```
>
..+....1....+....2....+....3....+....4....+....5....+....6....+....7.00023
8 000183          C1-VALIDATE-DATA.
000239 000184          *-----*
000240 000185
000241 000186          IF IN-RATE-OF-PAY NOT > 0    OR  IN-RATE-OF-PAY
000242 000187          1      IF IN-HOURS NOT > 0    OR  IN-HOURS > 45
000243 000188          2          MOVE MULTIPLE-MESSAGE TO OUT-MESSAGE
000244 000189          1      ELSE
000245 000190          2          MOVE RATE-OF-PAY-MESSAGE TO OUT-MESSAGE
000246 000191          1      END-IF
000247 000192          ELSE
000248 000193          1      IF IN-HOURS NOT > 0    OR  IN-HOURS > 45
000249 000194          2          MOVE HOURS-MESSAGE TO OUT-MESSAGE
000250 000195          1      END-IF
000251 000196          END-IF.
```

Once the abending line is established it is sometimes possible to spot the reason for the error; it may be an obvious logic problem. You may have to carry out a **dry run** by working through the logic to derive the error. A dry run is where you imitate the computer processing a sample record, and follow the record through the processing step by step. This is best used when you know which input record was being processed at the time of the abend.

However, if this does not throw any light on the problem we may want to look within memory to establish the contents of various fields at the time of the dump. It may be that they contain data which is invalid in some way or in an incorrect format, or if not may indicate which input record was being processed at the time of the abend. The next section will cover the basic techniques to establish the contents of the fields.

## Finding the Contents of Fields

We will use the same example dump as in the previous section, where an OC7 has been identified and the COBOL instruction found to be

```
IF IN-HOURS NOT > 0 OR IN-HOURS > 45
```

The OC7 occurring on that line, tells us that one of the fields referenced in that line contains invalid data for its type, so the next stage is to find out the contents of those fields, so we can ascertain how they came to contain invalid data.

First, note down the name(s) of the fields which occur in the abending COBOL statement.

We then need to establish where these fields are located within memory (i.e. their hex. address within the dump of main storage). To do this we need to use the **Data Division Map**.

This is a table which contains a listing of all the files, records, groups and individual fields defined within the program. The names are listed down the left-hand side in the order in which they are defined in the program, together with their definition level. (There is a lot of other information in this table, and further explanations can be found in the Compilation Listing tutorial notes.)

The columns we are interested in are **Name, Base Locator, Displacement Blk** and **Data Type**. Look down the left hand side of the table and find the field names that you are interested in. Then note down the value of the associated Base Locator field together with the value in the Displacement column and the format and length of the field. In the tutorial example we have field IN-HOURS on the abending line. In the table the values are

Name	Base Locator	Disp Blk	Data Definition	Data Type
01 INPUT-FILE . . . . .	BLF=0000	000	DS 0CL20	GROUP
02 IN-PAY-NO . . . . .	BLF=0000	000	DS 5C	DISPLAY-NUM
02 IN-SEX . . . . .	BLF=0000	005	DS 1C	DISPLAY
02 IN-HOURS . . . . .	BLF=0000	006	DS 2C	DISPLAY-NUM.
.....				

## Base Locators

Each field in the program can be found at a different address within the dump. In order to interrogate the contents of the field we have to calculate its hex. address.

**Each** file defined in a program is located in memory at a hexadecimal absolute address. The first field in this file begins at this byte with all other fields following consecutively in memory at displacements from this address. For example, if the fields on file are

```

01 INPUT-FILE
    02 PAYROLL-NO          PICX(4).
    02 NAME                PICX(20).
    02 DATE-OF-BIRTH      PIC 9 (6).
etc.

```

and the first byte of the file was at 00002137 hex. in memory then PAYROLL-NO would be at hex. displacement 0, NAME at hex. displacement of 4 and so on. The address of each field could be calculated by adding the start address of the file to the displacement of the field in question. Both values are in hex. so there should be no problem. For example, the address of DATE-OF-BIRTH in the dump is

$$00002137 + 24 \text{ } 0000215B$$

The address of all fields within files can be calculated in this way.

Similarly all working fields are found at hex. displacement from the beginning of Working Storage in the dump, and this address can be calculated in the same way.

Unfortunately the start address of each file and Working Storage are not immediately obvious to the programmer - they do not appear in the Data Division Map.

The start address for each file is placed in a separate **BASE LOCATOR**. These Base Locators are numbered 0, 1, 2 etc.; Base Locator 0 is associated with the first file defined in the program, 1 for the second and so on. These are known as BASE LOCATORS for Files or **BLF's** .

## **The VS COBOL II Abend Information File**

Working Storage usually has one Base Locator associated with it, known as a BASE LOCATOR for Working Storage or BLW. (Sometimes there may be more numbered 0, 1 etc. depending on the size of Working Storage).

The Data Division Map only tells us which type and number Base Locator is associated with each file/storage and each fields displacement from the start address denoted by the Base Locator. In order to find the absolute address held by the base locator we have to look in another file within the job - usually the last one. It is entitled VS COBOL II ABEND INFORMATION. This extra output file will only be produced in your job if you code a

```
//SYSABOUT      DD SYSOUT=*
```

statement in your JCL, (unless you are using the TEST catproc as this statement is included within the catproc)

The output in this file may look something like this

```
--- VS COBOL II ABEND Information ---  
Program = 'TRDUMP' compiled on '02/09/00' at '15:03:16'  
  TGT = '0DC21028'  
Contents of base locators for files are:  
  0-00028FB0      1-0DC21521  
Contents of base locators for working storage are:  
  0-0DC215C0  
Contents of base locators for the linkage section are:  
  0-00000000  
No variably located areas were used in this program.  
No EXTERNAL data was used in this program.  
No indexes were used in this program.  
--- End of VS COBOL II ABEND Information ---
```

### ***Calculating the Address of the Field***

The contents of each base locator is in the form of the BL number then a hyphen and then an 8 digit hex. address. This address is the absolute address of the file or working storage associated with that particular base locator.

In our example we know that the field IN-HOURS is associated with BLF 0 and that from the COBOL II Abend Information we can see that the input file begins at an absolute address of 00028FB0.

We know that IN-HOURS is located at a displacement of hex. 006 from the start of the file, therefore we can calculate the absolute address of IN-HOURS as being

$$00028FB0 + 006 = 00028FB6$$

Note that this procedure of calculating the address of the field in storage would have to be repeated for each file on the abending line.

## Interrogating the Dump

The next step is to find the field within the dump to interrogate its contents. The dump can usually be found in file 6. At the top of the dump there is a repeat of the dump statistics found in file 1. This is followed by a number of internal tables, save areas and register contents, the use and significance of which is covered in the Advanced Dump Solving tutorial. The area of the dump we are most interested in at this stage begins where it says ACTIVE LOAD MODULES. (We can locate this by using the NEXT or FIRST ROSCOE command).

Down the left hand side of the screen is a series of hex. addresses. To the right of this are 8 four byte columns of hexadecimal data - this is the actual data stored within memory. (Remember two digits represent one byte of information; or one digit/character.) On the right hand side of the page is the EBCDIC representation of the data.

```
ACTIVE LOAD MODULES
LPA/JPA MODULE

NAME=IGZCLNK
00006220 47F0F034 2740C9C7 E9C3D3D5 D2404200      F3F2F44B F0F0F161 F0F661F9 F340F1F9
00006240 4BF2F940 4040D500 0000624C F0F0F0F0      F0F0F040 90ECD00C 05C01841 189D9102
00006260 40004780 C0E458A0 905858B0 90E812BB      4780C02A 18DB1F00 5000D048 9200D150
00006280 47F0C250 1BEE58FA 004041E0 F1AC58FE      000012FF 4780C0B6 4110019A 4111003F
000062A0 5410CB76 5910F000 4720C0B6 41000020      1B101AF1 50F0E000 18EF1BE1 1801D20F
```

.....

In order to find the line in which you are interested you can use the NEXT command to search for the address as a character string. The hex. addresses increment by 20 hex on each line, therefore you have to search for address in multiples of 20. For example, if you required a field at address 000257B3 you could use

NEXT /257A0/ or

NEXT /257C0/

to search for the closest address. In our example we would use 'NEXT /28FA0/'

Once you have located the line you count across the bytes for the address you want, starting with zero. For example, for a field at address 0003566A we would search for address 35660 and then count across to the 11th byte. We will already know the format and length of the field from the Data Division Map and should be able to examine the field contents. In our example we know that the field begins at 00028FB6

```
00028FA0 F3F6F5F0 40F0F5F5 F040D440 40404040 F1F2F2F3 F3D44040 0703682C F6F3F7F8
```

and is defined as 2 bytes numeric but the 2 bytes starting from address 00028FB6 contain spaces ("4040"). Therefore this is the error.

## Appendix A

Abend Code	Description	Possible Causes
12	QSAM Error No dump	Not strictly an abend. READ after End Of File; after a CLOSE or before the file was OPENED. Missing DD statement.
001	I/O error  Msg gives file concerned	BLOCK CONTAINS 0 omitted, BLOCK CONTAINS X with non-zero X. WRITE after a CLOSE Conflicting DCB information.
002	I/O error Msg gives file concerned	Too large a variable length record: . input - RDW corrupted . output - implied RDW > LRECL
013	OPEN failure Msg gives file(s) in error	Conflicting DCB info. between file, program and JCL. Member not found.
0C1/2	Invalid Operation	Gone to wrong part of core. Incorrectly linked program (e.g. missing subroutine). Invalid CALL statement.
0C4	Storage Protection	Subscript/index out of range of table. Incorrectly linked program. Moving data to FILE SECTION area of unopened or incorrectly defined file.

<b>Abend Code</b>	<b>Description</b>	<b>Possible Causes</b>
0C7	Data Exception	Most common dump, with 1001 reasons including: - Attempt to perform arithmetic on invalid packed decimal field. - Uninitialised field. - Usage in program does not match file. - MOVE to group 01 level field.
122/222	Operator Cancel	Excessive CPU lines.
322	CPU time exceeded	Loop in program, or too small a TIME parameter on the jobcard.
722	SYSOUT exceeded	Loop in program.
806	Load module not found	Check compile/link steps - possible completion code is > 0.
B37	Run out of disk space	Not enough space allocated. Not enough space on disc. Possible program loop.

QSAM ERROR Followed by a number of possible status codes:

- 39 - conflict of fixed file attributes.
- 41 - OPEN attempted for a file already open.
- 42 - CLOSE attempted for a non open file.
- 44 - attempt to write a sequential file record with a record of a different size.
- 46 - sequential READ attempted with no valid next record.
- 90 - Other errors with no further information.



92 - Logic error