Session 13039

z/OS Debugging: *Everything you need to know to shoot an 0C4*

MVS Core Technologies Project – February 5th, 2013

Evan Haruta haruta@us.ibm.com
Patty Little plittle@us.ibm.com

IBM Poughkeepsie

Trademarks

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

- **MVS**
- **OS/390®**
- **z/Architecture®**
- **z/OS®**

* Registered trademarks of IBM Corporation

Table of Contents

What is an 0C4?

- A **completion code** is a 3-digit code used by the system to indicate the reason for abnormal termination of a unit of work after an error
- A non-resolvable **program interrupt** (commonly called a **program check**) can result in completion codes of ABEND0Cx, ABEND0Dx or ABEND0E0
- An **ABEND0C4** completion code may be issued after the following program interrupt codes (PIC):
  - **PIC 4, 10, 11, 38, 39, 3A, 3B**
  - The reason code is the PIC

The term 0c4 actually comes from the completion code of ABEND0C4. A completion code is used by the operating system to indicate the error that caused the abnormal termination of a program. A non-resolvable program interrupt is one that cannot be resolved by the operating system, for example, a page fault caused by a program accessing storage that has not been getmain'd. The completion codes of ABEND0Cx, ABEND0Dx and ABEND0E0 are dependent on the program interrupt code. An ABEND0C4 can occur after a PIC 4, 10, 11, 38, 39, 3A and 3B. The Principles of Operation documents what these program interrupts are.

0C4 completion codes in a dump

- IPCS **ST FAILDATA** or **VERBX LOGDATA**
  - Error information in the dump header or a LOGREC entry
- IPCS **SYSTRACE**
  - RCVY system trace entries
- IPCS **SUMM FORMAT**
  - In the TCB or the TCB Summary
  - In the RTM2WA (if available)

For an SVC dump taken for an ABEND0C4, IPCS ST FAILDATA shows the completion code.

For any SVC or S/A dumps, you can find ABEND0C4 completions codes in the LOGRECs, system trace, TCB or RTM2WA..

Program interrupt processing

When a program interrupt occurs :

- **Hardware** (the mainframe computer)
  - Updates PSA with
    - ILC/IC (Instruction-Length Code/Interrupt Code)
    - TEA (Translation Exception Address), if applicable
  - Gives control to z/OS via a PSW swap
- **Software** (the z/OS operating system)
  - If resolvable, handles interrupt and then resumes interrupted program
  - If non-resolvable, terminates current unit of work with completion code

To debug an ABEND0C4, there are 3 important pieces of information: (1) the Instruction Length Code (ILC) (2) the Interrupt Code (IC) and (3) the Translation Exception Address (TEA). The TEA is only applicable to some of the program interrupts, to be discussed on future slides in this presentation.

ILC, IC and TEA

- **ILC** (Instruction Length Code)
  - The length of the instruction that caused the program interrupt
- **IC** (Interrupt Code)
  - The program interrupt code (commonly known as PIC)
- **TEA** (Translation Exception Address)
  - Contains the virtual address that caused a PIC 10, 11, 38, 39, 3A or 3B, significant to the page boundary

The ILC, IC and TEA enable you to investigate the following:

- what was the failing instruction?
- what was the virtual address related to the program check?
- did the error occur while accessing an operand or fetching an instruction?

Translation exception address (TEA)

- Contains first portion of the virtual address causing a PIC 10, 11, 38, 39, 3A, or 3B, and bits describing the cross-memory environment at the time of the program interrupt
- Use it to determine why the error occurred, and whether the error occurred **during operand access** or **instruction fetch**

z/Architecture 8-byte TEA

virtual address	
0 – 51	52 – 63

(stored after PIC 10,11,38,39,3A,3B)

```
Bit 0-51 0-51 of virtual address
  52-59 Unpredictable
  62-63 00 Primary ASCE
        01 AR Mode
        10 Secondary ASCE
        11 Home ASCE
```


The Translation Exception Address is stored in the PSA by hardware when certain program interrupts occur. Since a virtual address can be 64-bit in z/Architecture, the TEA has been expanded to 2 words. Another common name for the TEA is Translation Exception ID (TEID). Note that the TEA in the PSA is usually reused by the time the dump is taken, so the best places to find the TEA for the error in question are ST FAILDATA, RTM2WA, system trace and LOGREC.

The TEA also contains information after program interrupts related to Cross-Memory Access and Protection Exceptions. See z/OS Principles of Operations manual for details.

ABEND0C4 causes (architectural view)

- **Non-resolvable** segment fault (PIC 10)
- **Non-resolvable** page fault (PIC 11)
- **Non-resolvable** ASCE fault (PIC 38)
- **Non-resolvable** region fault (PIC 39, 3A or 3B)
- **Disabled** segment/page fault
- **Disabled** region/ASCE fault
- **Protection exception** (PIC 4)

There are many possible causes for an ABEND0C4. All the clues required to further investigate the cause of the abend0C4 can be found in the dump. The debugger should first locate the Program Interrupt Code (PIC) to understand WHY the abend occurred. The Principles of Operations manual gives very detailed explanations of each of these interrupt codes. Next, the debugger should look at the environment at the time of the abend.

ABEND0C4 causes (a simpler view)

- **Non-resolvable** PIC 10 or 11
  - An invalid below-the-bar virtual address was being used
- **Non-resolvable** PIC 38, 39, 3A or 3B
  - An invalid above-the-bar virtual address was being used
- **Disabled** PIC 10, 11, 38, 39, 3A or 3B
  - A page not backed by real storage was accessed by a program while it is disabled for certain interrupts (I/O and external)
  - Check the second digit of the failing PSW
    - If it is 4 the program is Disabled
    - If it is 7 the program is Enabled
- **Protection exception** (PIC 4)
  - A program violated a storage protection protocol

Note that a Disabled PSW is not allowed to take a program interrupt (such as a segment or page fault), even if the faults are otherwise resolvable. (The only exception to this is if the storage is DREF - Disabled Reference). In these cases, the root cause is either a bad storage address, or the program is running DISABLED in error.

PIC 38, 39, 3A and 3B are program interrupts related to storage access above the 2G bar.

IP STATUS FAILDATA or VERBX LOGDATA

```
Symptom               Description
-------               -----------
PIDS/5752SC100        Program id: 5752SC100
RIDS/IFAEDABC#L       Load module name: IFAEDABC
RIDS/IFAEDABC         Csect name: IFAEDABC
AB/S00C4              System abend code: 00C4
PRCS/00000010         Abend reason code: 00000010
REGS/C1016            Register/PSW difference for R0C:-1056
RIDS/IFAEDDEF#R       Recovery routine csect name: IFAEDDEF
```


IPCS STATUS FAILDATA contains all the necessary data required to begin debugging a program check. Debugging steps and examples will be shown later in the presentation.

IP STATUS FAILDATA or VERBX LOGDATA

```
TIME OF ERROR INFORMATION

PSW: 47044400 80000000 00000000 2747B016
Instruction length: 06   Interrupt code: 0010
Failing instruction text: B24D005C 1846D207 50104038
Translation exception address: 00000000_00810001

Breaking event address: 00000000_2747A6E0
Registers 0-7
GR: 00000003 2F16DB18 29E58510 069E2590  008100C2 2F089B20 006000C2 007FF6D8
AR: 00000000 00000000 00000002 00000000  00000000 00000000 00000000 00000002
Registers 8-15
GR: 2F16E618 00000000 2F16EDEE 2F16DDEF  2746E5B4 2F16CDF0 29E58510 00000000
AR: 00000000 00000002 00000000 00000000  00000000 00000000 00000000 00000000

Home ASID: 02C0    Primary ASID: 0011    Secondary ASID: 0011
```


Note the ILC, IC and TEA for this ABEND0C4, shown in the output of ST FAILDATA.

IP SUMMARY FORMAT ASID(x'nn')

```
TCB: 007FF6D8
    +0000  RBP...... 007FF650  PIE...... 00006E00  DEB...... 00000000
    +000C  TIO...... 007BDFE8  CMP...... 940C4000  TRN...... 40000000
    +0018  MSS...... 7F472928  PKF...... 80        FLGS..... 01000000
    +012C  EAE...... 7FFFE1D8  ARC...... 00000010
```

```
                          RTM2WA  SUMMARY
                         --------------
+001C  Completion code                    840C4000
+008C  Abending program name/SVRB address 007FF8E0 00000000
+0094  Abending program addr              00000000

       GPRs at time of error
     0-3   00000003  2F16DB18  29E58510  069E2590
      4-7  006000C2  2F089B20  008100C2  007FF6D8
      8-11 2F16E618  00000000  2F16EDEE  2F16DDEF
     12-15 2846E5B4  2F16CDF0  29E58510  00000000

+007C  EC PSW at time of error  470C4400 A747B016 00060010 00810001
```

ILC and IC

In the IPCS SUMMARY FORMAT display, you can find the ABEND TCB by checking the completion code. Note that the TCBCMP field may show residual data from a prior ABEND which has been successfully retried. If you page down through the display and see a corresponding RTM2WA, the TCBCMP field is not showing residual data, and can therefore be used during dump analysis.

The 64-bit TEA can be found in RTM2TRNE (+6C8 in the RTM2WA).

IP SYSTRACE ASID(x'nn')

ILC IC TEA

```
01 02C0 007FF6D8   PGM   010 470C4400 A747B016  00060010 00000000
                                                         00810001
01 02C0 007FF6D8  *RCVY PROG                    940C4000 00000010 00000000
```

completion code reason code

You can find the same data in the IPCS SYSTRACE display.

General diagnostic approach

- Gather program check data
  - LOGREC/LOGDATA, ST FAILDATA, and RTM2WA control blocks are good sources of information
- Analyze program check data
- Read compiled listing of source code, beginning at the failing PSW and moving backwards to understand why the flow led to this error
- Review LOGREC, system log and system trace for history of events leading up to this error
- Review SUMMARY FORMAT for chronology of module flow leading up to this error under failing TCB

Steps described in blue represent topics covered in this presentation. The remainder of the bullets represent logical diagnostic follow-on including code inspection and use of additional IPCS reports.

Program check data can be located in various places including LOGDATA (generated via the IPCS VERBX LOGDATA command), ST FAILDATA (which queries the SDWA), and the RTM2WA. System Trace, formatted via the IPCS SYSTRACE command, also shows some program check information. Where to look for program check data in any given dump depends on the error environment and the dump.

We will see in this presentation that a quick analysis of program check data will allow us to better understand the nature of the error, propose theories and define a logical next course of action in our debugging.

Program check analysis: Questions to consider

- What is the Program Interrupt Code (PIC)?
  - What does it tell us about the nature of the program check?
    - PIC 10, 11, 38, 39, 3A, 3B: Fault-related Translation Exception
    - PIC 4: Protection Exception
    - Consult z/OS Principles of Operation for more information
  - What does it tell us about our debugging approach?
    - **PIC 10, 11, 38, 39, 3A, 3B:**
      PSW points **DIRECTLY AT** failing instruction
    - **PIC4:**
      PSW points **IMMEDIATELY AFTER** the failing instruction
- What is the length of the failing instruction (ILC)?
  - Needed especially when PSW points AFTER failing instruction

The program interrupt code (PIC) that accompanies an ABEND0C4 further clarifies the nature of the program check. As debuggers, there is little need for us to distinguish between the various PICs that occur due to a translation exception. They all carry the same meaning – that a program tried to touch a storage address that for some reason was not available for it to touch. This could be because the address was that of storage that had been freed, or it could be because the address was invalid.

A protection exception occurs when a program tries to touch storage that it is not allowed to touch due to protection on that storage.

For program checks that occur as a result of a translation exception, the PSW at time of error will point directly at the failing instruction. For protection exceptions, the PSW will point immediately after the failing instruction. This means that, for a protection exception, we will need to back up the PSW to get to the failing instruction.

Where did the error occur?

- Locate the error PSW and Registers
  - IPCS BROWSE or do a WHERE on the PSW address
- Locate the failing instruction
  - Failing instruction text in LOGDATA, RTM2WA, etc
    OR
  - **IPCS BROWSE / LIST** failing instruction address
  - **For PIC4**, remember to back up PSW address by instruction length!
- What is the failing instruction?
  - **IP OPCODE** failing_instruction
    OR
  - **IP LIST** failing_instruction_address **I**

Program check PSW and registers can be found in LOGDATA, LOGREC, ST FAILDATA, and the RTM2WA. The program check PSW can also be found in system trace; however, the registers cannot be found there. Once the PSW address is obtained, map it to a module and offset. This can be done via the IPCS WHERE command in some cases. In other cases it may be necessary to use IPCS BROWSE to identify the module. It is likely that the debugger will need to examine the code in this module to successfully diagnose the program check.

The failing instruction text is 12 bytes of data that is collected by RTM as it handles the error. RTM takes the error PSW, and gathers 6 bytes of storage immediately before the PSW, as well as the 6 bytes of storage pointed to by the PSW. Since the maximum instruction length is 6 bytes, and since the error PSW will always point either immediately after or right at the failing instruction, the failing instruction text it guaranteed to include the failing instruction.

Examples: IP OPCODE ; IP LIST addr I

Provide instruction

IP OPCODE 58304000

Mnemonic for X'58304000' is L

Provide address of instruction in storage

IP LIST 20F54 I

LIST 020F54. ASID(X'00B2') LENGTH(X'04') INSTRUCTION
00020F54 | 5830 4000 | L R3,X'0'(,R4)

The failing instruction text will be a series of hexadecimal digits. We will need to convert the hex to a meaningful opcode. The IPCS OPCODE command can be used to translate a hex opcode to its mnemonic. You may enter just the opcode, or you may enter the whole instruction. Regardless of which you enter, IPCS will only return the mnemonic. IPCS LIST with the I (for Instruction) option will actually interpret the entire instruction. You must provide the address of the failing instruction.

Consider the failing instruction

- Gather PIC-specific information
  - PIC 10, 11, 38, 39, 3A, 3B: Locate Translation Exception Address (TEA)
  - PIC 4: Note PSW execution key
    - 078D0000 90234568 = Execution Key8
    - 07850000 80000000 00000000 10234568 = Execution Key8
- Identify assembler base & index registers of failing instruction
  - For fault-related PIC, which register matches Translation Exception Address?
    - Gotcha #1: TEA may match combined total of Index Register and GPR
    - Gotcha #2: TEA may match Register+Displacement or Register+Length of Move
    - Gotcha #3: TEA may match address of instruction being fetched
  - For PIC4:
    - Does register contain zero or pointer to PSA low core (0-1FF, 1000-11FF) ?
      OR
    - Does storage key of address in register not match PSW execution key ?
      IP LIST address **DISPLAY**

When debugging a program check for a Translation Exception, you will need to locate the Translation Exception Address (TEA) in the error data.

When debugging a protection exception, you will need to make note of the PSW key, which is the 3rd nibble of the PSW. You will likely also need to identify the storage key of the page being touched. This can be done with the IPCS LIST command with the DISPLAY option. Output of this command is demonstrated and interpreted on the next slide.

In important part of debugging a program check is understanding the format of the failing instruction. It is important to be able to identify the base register(s), index register(s), displacement(s), and length(s) that comprise the instruction.

A simple ABEND0C4 PIC11

From LOGREC (some lines omitted)

```
TYPE:  SOFTWARE RECORD       REPORT:  SOFTWARE EDIT REPORT           DAY.YEAR
JOBNAME: ABCD        SYSTEM NAME: SYSA
ERRORID: SEQ=00251  CPU=0080  ASID=0080  TIME=13:41:18.1

SEARCH ARGUMENT ABSTRACT
  AB/S00C4 PRCS/00000011 REGS/0E014 REGS/0C266

TIME OF ERROR INFORMATION

  PSW: 07041000 80000000 00000000 050A1856
  INSTRUCTION LENGTH: 06   INTERRUPT CODE: 0011
  FAILING INSTRUCTION TEXT: 00805850 B168D93F B2245000
  TRANSLATION EXCEPTION ADDRESS: 00000000_2D8D9800

  BREAKING EVENT ADDRESS: 00000000_050A20BC
  REGISTERS 0-7
  GR: 00000004 2D8D9FB8 2D8D9FC3 0000000C  00000008 2D8D9FB8 050A45ED 00FC8D00
  AR: 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
  REGISTERS 8-15
  GR: 050A35EE 00000000 050A25EF 2D824BB8  850A15F0 2D824D94 850A1842 00000080
  AR: 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
```

Interrupt Code = PIC11:
PSW points AT failing instr

Instruction Length = 6:
Failing instr is D93FB2245000

IP OPCODE D93FB2245000
MVCK instruction

Instruction base registers are
Reg11 (B) and **Reg5**

TEA = 2D8D9xxx matches Reg5

Conclusion: Storage pointed to by register 5 is not available.
This requires further investigation.

Now we’re going to debug some program checks. We’ll see that the data we gather for each is fairly consistent one to the next. However, the failing instruction and corresponding register content are the variables that result in a different interpretation for each ABEND0C4 example that we will be looking at. We will see that the execution environment of the PSW (addressing mode, cross memory mode) will also play a role. In this first example, we have a PIC 11 which is a form of translation exception. For a translation exception error, the PSW points at the failing instruction. We translate this instruction opcode to determine that it is a MVCK instruction. (If you encounter an instruction with a format that you are unfamiliar with, you can look it up in the Principles of Operation.) A MVCK instruction has two base registers, one that points to the source of the data being moved and the other that points to the target. Either of these registers could be the reason for the translation exception. Compare the TEA to the instruction base registers in order to determine which triggered the program check. We discover it matches register 5; for some reason the storage indicated by this register is not available to the program.

A simple ABEND0C4 PIC10

From ST FAILDATA (some lines omitted)

```
Time of Error Information

  PSW: 04040000 80000000 00000000 014CAB2A
  Instruction length: 04   Interrupt code: 0010
  Failing instruction text: 0010B20A 200098EF 1000B20A
  Translation exception address: 00000000_4F4F4000

  Breaking event address: 00000000_014CAAE8
  AR/GR 0-1     FFFFFFFF/00000000_00000058  00000000/00000000_4F4F4F57
  AR/GR 2-3     FFFFFFFF/00000000_0000040C  FFFFFFFF/00000000_81030A38
  AR/GR 4-5     FFFFFFFF/FFFFFFFF_0215BA28  FFFFFFFF/00000000_00000C00
  AR/GR 6-7     FFFFFFFF/00000000_04355C00  FFFFFFFF/00000000_4F4F4F4F
  AR/GR 8-9     00000000/00000000_04435B18  00000000/00000000_00000001
  AR/GR 10-11   00000000/000001EF_810303D8  00000000/000001EF_7FFFCA98
  AR/GR 12-13   00000000/00000000_814CAB0A  00000000/00000000_7FFFCA98
  AR/GR 14-15   00000000/00000000_0000030A  00000000/00000000_00000000
```

Interrupt Code = PIC10:
PSW points AT failing instr

Instruction Length = 4:
Failing instr is 98EF1000

IP OPCODE 98EF1000
LM instruction

Instruction base register is **Reg1**

TEA = 4F4F4xxx matches Reg1

Conclusion: Storage pointed to by register 1 is not available.
Register 1 does not look like an address.
Examine code to understand how R1 was derived.

In this example of a PIC 10, we gather the same information as we did for the previous PIC 11. This time the failing instruction is a LM, which has one base register. The content of register 1 at time of error matches the TEA. For some reason the program could not touch storage at this address. One possibility is that the storage has been freed. However, if you look at the content of register 1, you can see that it looks repetitious (similar to register 7 which is even more repetitious) which would suggest the problem is not with freemained storage, but rather with an invalid value in register 1 that was never meant to be interpreted as an address in the first place. The debugger will need to examine code to understand where this invalid address came from.

You may notice that the PSW for this error is disabled. (The second nibble = 4.) You may recall that we learned earlier that if disabled code suffers a fault, then an ABEND0C4 results. In this example, disabled code did indeed take a fault. However, the root of the problem is not that the code was disabled, but rather that the bogus address in register 1 triggered an invalid storage reference. When disabled code suffers a fault, the most common reason is because the translation exception address is invalid, as is the case here. However, other possibilities include that the storage in question was supposed to be fixed but wasn't, or that the code really should not have been disabled in the first place. Considering the register content and understanding the abending code are the keys to figuring out which is the case.

Translation Exceptions: Questions for consideration

- **Is storage address unreasonable?**
  - How was this address obtained? Was the source corrupted? Examine code, regs.
  - Could a bad branch have occurred leading to random instruction execution?
- **Is storage address "reasonable" such that it could have been valid at one time but freed?**
  - Local storage can be freed by other TCBs
  - Global storage can be freed by other address spaces
  - Subpool FREEMAINs can occur at task termination
- **Other considerations for a "reasonable" storage address**
  - Is the PSW disabled and the storage not fixed?
  - How was storage address obtained? (Examine code, registers)
  - Is addressing mode (AMODE) correct?
  - Is cross memory environment correct?

When debugging a Translation Exception, consideration must be given to the value in the register that triggered the exception. Sometimes the register content will be very obviously invalid as an address, instead containing an eyecatcher or pattern such as what we saw in our PIC10 example. In these cases, it is important to consider how this data was derived. Usually this means reading the abending module, starting at the point of error and working backwards. Occasionally a bad branch results in random instruction execution with equally random results that would typically be a program check with random register data. In this case, the trick is to understand the bad branch. This means playing with the registers at time of error, looking for pointers into code, as well as considering the Breaking Event Address (BEA) which is reported in places where other program check data lives.

Often a translation exception address appears reasonable. In cases like this, it could be that the storage in question has been freed. If the PSW is disabled, this could a case of disabled code touching storage that is not fixed. Other possibilities include "gotchas" such as addressing mode and cross memory mode considerations that influence what virtual address the failing instruction was referencing. We look at this more in subsequent examples.

Freemained storage is the most common reason for a translation exception on an apparently reasonable address. One of the first steps in diagnosing such a program check is to look for freemain activity for that address (and its vicinity) in the system trace table. When doing this analysis for a local (private) storage address, remember to consider the possibility of a subpool freemain, which frees storage implicitly by subpool rather than explicitly by storage address. When trying to locate the freemain of a global storage address, remember that this could come from any address space.

Protection exception (PIC 4)

Most common causes:

- **Key-controlled protection**
  - PSW execution key must either match storage key or be KEY0 when:
    - Writing to storage
    - Reading **fetch-protected** storage
  - Otherwise, an ABEND0C4 PIC 4 will occur

- Low core address protection
  - PSA x'000-1FF' and x'1000-11FF' are **write-protected**

A protection exception can be caused by violating the common protection mechanisms. Storage access is protected by matching the PSW execution key to a page's storage key. If a task is running PSW key0, it will have access to storage of any storage key. The next slide shows how to determine a page's storage key.

Note that updates to storage are always key-protected. A program must be executing either in the key of the storage it is touching or in key0 in order to successfully update storage.

Even programs running execution key0 cannot update PSA locations 0-1FF and 1000-1FFF.

Note that storage can be defined as fetch-protected, which means that it can only be referenced either by programs running with a matching execution key or running key0.

How to determine the storage key

- Associated with 4K (1 Page) of storage

KKKK	FRC0
0	4 7

KKKK	=	Key (4 bits)
F	=	Fetch-Protection Bit
R	=	Reference Bit
C	=	Change Bit
0	=	Reserved

- To find the storage key in a dump:
  - LIST storage_addr DISPLAY

```
LIST 0097F5E0 CPU(X'00) ASID(X'0001') LENGTH(4) AREA
CPU(X'00) ASID(X'0001') ADDRESS(0097F5E0) KEY(06) ABSOLUTE(01CDB5E0)
0097F5E0.  00000000
```

Key 0

x'6'= b'0110'
Not fetch-protected

There is a storage key associated with each page of storage. Programmatically, the storage key can be accessed through instructions such as SSKE and ISKE. If you are debugging a protection exception (PIC 4) and would like to determine the storage key of an address in a dump, issue IP LIST storage-addr DISPLAY and check the first nibble of the KEY value.

This output also includes the fetch-protect status of the page, which is reflected in the first bit of the second nibble of the KEY(xx) output. If the fetch-protect bit is on, then the page is fetch-protected.

Less common protection exception

- **Page protection**
  - A page is write-protected if the **page protection bit** is ON in the page table entry
    - Verify via RSMDATA VIRTPAGE or RSMDATA HIGHVIRT report
    - Read-only nucleus and LPA have this bit turned on
  - Running DAT-off bypasses this protection
- When PIC 4 is due to page protection, TEA will be stored with bit 61 turned on (z/Architecture mode)
  - TEA will contain first portion of virtual address causing PIC 4

A full page can be protected by using RSM (Real Storage Manager) page services to turn on the page protection bit in the page table entry. Some areas in low core (PSA) are protected from WRITE.

See MVS Diagnosis: Reference for details about RSMDATA reports.

While the TEA is primarily for use with Translation Exceptions, there is one case where it is relevant for an ABEND0C4 PIC4. If a protection exception occurs because a program tries to store into a write-protected page, the TEA will be relevant as indicated by bit 61 being on.

A simple ABEND0C4 PIC4

From ST FAILDATA (some lines omitted)

```
Time of Error Information

 PSW: 07141000 80000000 00000000 3880A5A0
 Instruction length: 06   Interrupt code: 0004
 Failing instruction text: D5006008 3000A784 00084160

 Breaking event address: 00000000_3880A5A8
 AR/GR 0-1    00000000/00000000_00000000  00000000/00000000_00000001
 AR/GR 2-3    00000000/20000000_3893990B  00000000/00000000_388DCF90
 AR/GR 4-5    00000000/00000000_3870A07C  00000000/00000000_00000000
 AR/GR 6-7    00000000/00000000_00000930  00000000/00000000_3893990A
 AR/GR 8-9    00000000/00000000_3870A010  00000000/00000000_0003922A
 AR/GR 10-11  00000000/00000000_388D52B0  00000000/00000000_00007000
 AR/GR 12-13  00000000/00000000_3880A530  00000000/00000000_38939688
 AR/GR 14-15  00000000/00000000_B79C70A8  00000000/00000000_FFFFFFFD

 Home ASID: 002B    Primary ASID: 002B    Secondary ASID: 002B
```

Interrupt Code = PIC4:
PSW points AFTER failing instr

Instruction Length = 6:
Failing instr is D50060083000

IP OPCODE D50060083000
CLC instruction

Instruction base registers are
Reg6 and **Reg3**

PSW Key = 1

Note: To update storage or to reference fetch-protected storage, PSW key must match the page key or else be KEY0. Need to verify page keys and fetch-protect status.

Here we are looking at an ABEND0C4 PIC4 protection exception. We gather PIC, ILC, and failing instruction, and we identify instruction base registers just as we did with the translation exceptions. However, in the case of a PIC4, we must remember to back up the PSW by the instruction length to get to the correct failing instruction.

Once we examine the failing instruction to identify the register and storage address drawing the protection exception, we must consider why this storage was protected from this program. To do this, we need to compare the PSW execution key and the storage key/fetch protect status to understand why hardware detected a violation. In this example, the PSW key is 1 while the storage key is 0 for the storage pointed to by base register 6, as we will see on the next slide.

Debugging an ABEND0C4 PIC4 (cont)

Note: Source #1 R3 = 388DCF90   Source #2 R6 = 00000930

LIST 388DCF90 DISPLAY

```
LIST 388DCF90. ASID(X'002B') LENGTH(X'04') AREA
ASID(X'002B') ADDRESS(388DCF90.) KEY(18) ABSOLUTE(06_3D60FF90.)
388DCF90. 0190E080
```

Key 1
Fetch-protected

LIST 930 DISPLAY

```
LIST 0930. ASID(X'002B') LENGTH(X'04') AREA
ASID(X'002B') ADDRESS(0930.) KEY(08) PREFIXED
00000930. 00000000
```

Key 0
Fetch-protected

Conclusion: PSW Key1 can neither update nor reference the Fetch-Protected Key0 storage pointed to by register 6. Does R6 contain a valid address?

IPCS LIST with the DISPLAY option allows us to easily identify a page's key and fetch-protect status.

Another simple ABEND0C4 PIC4

From ST FAILDATA (some lines omitted)

```
Time of Error Information

  PSW: 07041000 80000000 00000000 014A11E4
  Instruction length: 04   Interrupt code: 0004
  Failing instruction text: D0BC5020 3000A7F4 00091872

  Breaking event address: 00000000_014A1084
  Registers 0-7
  GR: 000005D8 7F5D7B68 000A05D8 00000000  7FF15000 7FF15080 00000004 00002000
  AR: 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
  Registers 8-15
  GR: 7F5D7B68 014A1E40 7F4643D0 7F4AFAA8  00000000 7FF15438 814B7B4A 7FF15538
  AR: 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000001

  Home ASID: 036F    Primary ASID: 036F    Secondary ASID: 036F
```

Interrupt Code = PIC4:
PSW points AFTER failing instr

Instruction Length = 4:
Failing instr is 50203000

IP OPCODE 50203000
ST instruction

Instruction base register is **Reg3**

Reg3 = 0

Conclusion: Protection exception occurred due to attempting to store into protected storage in low core. Even code running KEY0 cannot store here. Investigate how R3 became 0.

Here is another flavor of ABEND0C4 PIC4. In this case, we don't need to look at the PSW execution key. When we identify the failing instruction, we see that it is a Store off of register 3. Register 3 contains 0 which means that the instruction is attempting to store into low core location 0. This is specially protected such that no one can store into this location. This includes key0 programs such as this one. (Note: The key is in nibble 3 of the PSW.)

Protection Exceptions: Questions for consideration

- Is storage address correct?
- Does storage live in desired key?
- Is PSW executing in desired key?
- Is addressing mode (AMODE) correct?
- Is cross memory environment correct?

When analyzing a protection exception, the questions to be considered are focused on the PSW's execution key and the storage key. However, just as with translation exceptions, a protection exception could result from use of a bogus storage address, or from an unexpected addressing mode / cross memory mode environment.

Beware of the Addressing Mode!

- Code can execute in any of 3 addressing modes
  - Amode24 – PSW address & registers interpreted as 24-bit values
    - PSW bits 31 and 32 = **00**; e.g. 070C000**0** **0**0DEF880
    - Ex: Reg1 = 12345678_9A**BCDEF0** ; storage accessed = **BCDEF0**
  - Amode31 – PSW address & registers interpreted as 31-bit values
    - PSW bits 31 and 32 = **01**; e.g. 078D200**0** **8**2BD3010
    - Ex: Reg2 = 12345678_**9ABCDEF0** ; storage accessed = **1ABCDEF0**
  - Amode64 – PSW address & registers interpreted as 64-bit values
    - PSW bits 31 and 32 = **11**; e.g. 077C100**1** **A**25F144C
    - Ex: Reg2 = **12345678_9ABCDEF0** ; all 64 bits used for storage access

- ABEND0C4 can result due to code in incorrect addressing mode

Addressing mode (AMODE) controls how many bits of the registers and PSW address get used for address translation. Program checks sometimes occur as a result of programs running in an incorrect AMODE. For example, a program that should run 24-bit mode but is running 31-bit mode may incorrectly use “dirty” data in the high order byte of a register, interpreting it as part of an address. A program that should run 31-bit mode may incorrectly truncate significant data in the high order byte of a register. In addition to causing program checks, both of these situations can cause overlays.

An ABEND0C4 PIC11 with a twist!

From ST FAILDATA (some lines omitted)

```
TIME OF ERROR INFORMATION

 PSW: 07041000 00000000 00000000 00FF0E24
 INSTRUCTION LENGTH: 04   INTERRUPT CODE: 0011
 FAILING INSTRUCTION TEXT: 60009608 B0585810 60005010
 TRANSLATION EXCEPTION ADDRESS: 00000000_00250800

 BREAKING EVENT ADDRESS: 00000000_01471B2C
 AR/GR 0-1     00000000/00000000_00000000  00000000/00000000_7F2503DC
 AR/GR 2-3     00000000/00000000_7F1D8138  00000000/00000001_00FDBC98
 AR/GR 4-5     00000000/00000000_008C57F8  00000000/00000000_008B9840
 AR/GR 6-7     00000000/00000000_7F2503DC  00000000/00000000_00F7E480
 AR/GR 8-9     00000000/00000000_01473E48  00000000/00000000_00000000
 AR/GR 10-11   00000002/00000000_7F1E7A18  00000000/00000000_7FFFBAF8
 AR/GR 12-13   00000000/00000000_00000080  00000000/00000000_7F1E23B8
 AR/GR 14-15   00000000/00000000_00FF0F00  00000000/00000000_AE71FB90

 HOME ASID: 006E    PRIMARY ASID: 006E    SECONDARY ASID: 006E
```

Interrupt Code = PIC11:
PSW points AT failing instr

Instruction Length = 4:
Failing instr is 58106000

IP OPCODE 58106000
L instruction

Instruction base register is **Reg6**

Reg6 = 7F2503DC
AMODE = 24 (PSW bits=00)
TEA = 00250xxx

Conclusion: Content of register 6 is interpreted as a 24-bit address rather than a 31-bit address since PSW is AMODE24. Should code be executing AMODE31 instead?

Here is another PIC11. We go through the usual steps in data gathering and identify that our failing instruction was a Load with base register 6 containing a value of 7F2503DC. However, notice that the TEA is 00250xxx, not 7F250xxx. The high order byte of the TEA is 00 whereas the high order byte of register 6 is x'7F'. The discrepancy is due to the PSW AMODE. Bit 31=0 and bit 32=0 which means AMODE=00 which is 24-bit mode. When running 24-bit mode, only the low order 3 bytes of register 6 will be considered in the address translation. The problem here likely is that the program should be executing in AMODE31 instead of AMODE24, although it is possible that someone obtained storage in the wrong location (above the line rather than below the line) instead.

A few words about Cross Memory

- Cross memory (XMEM) mode is the ability for a program to have addressability to multiple address spaces simultaneously through the use of Home, Primary, and Secondary address space definitions
  - Home – address space in which a program first begins executing
  - Primary – address space where program is presently executing**
  - Secondary – typically the address space where the program was most recently executing prior to switching to the current primary
- The most common way for a program to alter its cross memory environment is through execution of space-switching PC/PR instructions
  - PC instruction allows a program to "jump" to new code in another address space, thereby altering the primary address space
  - PR instruction restores PSW, registers, and cross memory environment to what existed at the time of the PC

**** Except when PSW ASC mode is Home. PSW ASC mode will be discussed shortly.**

When debugging program checks (and other abends) you need to be cognizant of the cross memory environment. Addressing storage at the right address but in the wrong address space is another reason that a translation exception or protection exception may occur.

Exploiting a cross memory environment

- Program PC/PR activity determines its Home, Primary, and Secondary address spaces
  - Instruction fetch typically from primary
  - Address space from which data is fetched is determined by PSW ASC mode bits (bits 16 and 17)
    - 00 – Primary
    - 10 – Secondary
    - 11 – Home
  - Program issues SAC instruction to change PSW ASC mode bits

- When debugging program checks, be aware of cross memory (XMEM) environment

Programs can enter cross memory environments through the issuance of space-switching PCs. Once in a cross memory environment, cross memory capabilities can be exploited through use of certain instructions such as SAC, MVCP, and MVCP.

An ABEND0C4 PIC4 with a twist!

From ST FAILDATA (some lines omitted)

```
Time of Error Information

 PSW: 07749001 80000000 00000000 291B9820
 Instruction length: 06   Interrupt code: 0004
 Failing instruction text: D2FFE000 F000B90A 00F0ECE0

 Breaking event address: 00000000_291B9824
 AR/GR 0-1     00000000/00000000_00000100  00000002/00000000_78CCCEBF
 AR/GR 2-3     00000000/00000000_0000007A  00000000/00000000_79E119D8
 AR/GR 4-5     00000000/00000000_78CBD000  00000000/00000201_6CCD5B28
 AR/GR 6-7     00000000/00000000_244CB240  00000002/00000048_0000FF15
 AR/GR 8-9     00000000/00000000_0000FF15  00000000/00000048_B02A1B48
 AR/GR 10-11   00000000/00000000_00000004  00000000/00000000_291C0840
 AR/GR 12-13   00000000/00000000_79E12367  00000000/00000000_79E11368
 AR/GR 14-15   00000000/00000000_78CC1FAA  00000000/00000048_B02A6A48

 Home ASID: 00B2    Primary ASID: 0061    Secondary ASID: 00B2
```

NOTE: Protection Exception on MVC of X'100' bytes. Program is running AMODE64 and KEY7. It is also running in Secondary ASC mode, meaning data access will be to the Secondary address space. SASID=B2. Need to check storage keys and fetch protect status.

Interrupt Code = PIC4:
PSW points AFTER failing instr

Instruction Length = 6:
Failing instr is D2FFE000F000

IP OPCODE D2FFE000F000
MVC instruction

Instruction base registers are **Reg14** and **Reg15**

HASID=SASID=B2
PASID=61
PSW ASC bits=10 (9=B'1001')
Secondary Mode

PSW Key = 7

Here we are looking at an ABEND0C4 PIC4 on a MVC instruction. Base registers are register 14 and register 15. We see that the PSW execution key is 7. We also note that we have a cross memory environment. Home and Secondary are ASID B2. Primary is 61. The PSW ASC mode bits indicate secondary mode, which means that data access will be to the secondary address space. (Instruction fetch will be from primary as is typical.) With this in mind, let's check the storage keys and fetch protect status of the storage indicated by registers 14 and 15. We see these checks on the next slide.

An ABEND0C4 PIC4 with twist (cont)

Note: Instruction = D2FFE000F000
R15 = 00000048_B02A6A48   R14 = 00000000_78CC1FAA
H=S=B2   P=61   PSW is **Key7**, Amode64, Secondary ASC

LIST 48_B02A6A48 ASID(x'B2') DISPLAY

LIST 48_B02A6A48. ASID(X'00B2') LENGTH(X'04') AREA
ASID(X'00B2') ADDRESS(48_B02A6A48.) **KEY(78)**

Key 7
Fetch-protected

LIST 78CC1FAA ASID(x'B2') DISPLAY

LIST 78CC1FAA. ASID(X'00B2') LENGTH(X'04') AREA
ASID(X'00B2') ADDRESS(78CC1FAA.) **KEY(78)**

Key 7
Fetch-protected

Conclusion: That's strange. The key of the pages pointed to by R15 and R14 match the PSW key. We were even careful and made sure we were looking at storage in the correct address space and using the correct AMODE. Why did we get a protection exception?
Hint: Check the MVC instruction length!

We repeat the relevant data from the previous slide at the top of this one. We display the key of the storage indicated by register 14 and by register 15 … both indicate Key7 which matches the PSW execution key. Bearing in mind that we are in a cross memory environment, we double check that we are looking at storage in the correct address space. We are. So why did we suffer a protection exception. Go back and look at the entire instruction. It is moving X'100' (=X'FF'+1) bytes of data from storage pointed to by register 15 to storage pointed to by register 14. Notice that the address in register 14 is very close to the end boundary (X'FFF') of the page. Adding X'100' to register 14 gives us address X'78CC20AA' which crosses us into the next page. This means that, when examining storage keys, we need to consider both the page pointed to by register 14 and the following page as well. We see the results on the next slide.

Debugging an ABEND0C4 PIC4 (cont)

Note: Instruction = D2**FF**E000F000 **Length** of data being moved = X'FF'+1 = X'100'
R15 = 00000048_B02A6A48 R14 = 00000000_78CC1FAA
H=S=B2 P=61 PSW is **Key7**, Amode64, Secondary ASC

Adding X'100' to register 14 causes us to cross into the next page!
We also need to consider the key of the page at 78CC2000 in ASID X'B2'.

LIST 78CC2000 ASID(x'B2') DISPLAY

LIST 78CC2000. ASID(X'00B2') LENGTH(X'04') AREA
ASID(X'00B2') ADDRESS(78CC2000.) **KEY(10)**

Key 1 (aha!)
Not fetch-protected

Conclusion: The protection exception occurred when the MVC instruction tried to move data onto page 78CC2000 which is KEY1 rather than KEY7. Was this storage obtained in the wrong key? Or perhaps the length specified on the MVC was too large? Or perhaps the storage pointed to by R14 was GETMAINed with too small of a size specified?

It turns out that the next page of storage is key1 rather than key7. This is why the protection exception occurred. This analysis opens to the door to several possible theories. It could be that the storage on page 78CC2000 was obtained in the wrong key. Or it could be that the length of the MVC was too long. (This can sometimes happen when there is a maintenance mismatch such that the code that does the MVC was compiled with the expectation that the data in question was X'100' bytes in length, while the code that obtained this storage was compiled with the expectation that the data area was significantly shorter.)

A word about Access Register Mode

- Like cross memory mode, access register (AR) mode allows a program to have additional addressability
  - Address space
  - Data spaces (data only spaces with max size of 2Gig)

- A program exploits access register mode by:
  - Gaining permission to access an address space or data space via an assigned "key" called an ALET. Some ALETs are predefined:
    - ALET = 00000000 can be used to access the primary address space
    - ALET = 00000001 can be used to access the secondary address space
    - ALET = 00000002 can be used to access the home address space
  - Placing the ALET in the access register that corresponds with the base register(s) of the assembler instruction that is accessing data
  - Issuing SAC to AR Mode to set PSW ASC mode bits = X'01'

Access Register (AR) mode is a step up from cross memory mode, allowing a program addressability to even more spaces simultaneously. Programs running in AR mode can access storage in other address spaces as well as in data spaces. (Of course these programs need to pass authorization checks to get this capability.) Debuggers need to be cognizant of AR mode environments when debugging program checks and other abends.

An ABEND0C4 PIC10: AR Mode

From ST FAILDATA (some lines omitted)

```
PSW: 07045001 80000000 00000000 0B0BBEA8
Instruction length: 06   Interrupt code: 0010
Failing instruction text: 00BCB904 00BAD503 B000C02B
Translation exception address: 00000008_00300801
Exception access identification: 0B

Breaking event address: 00000000_0B0BBE78
AR/GR 0-1     00000BB8/00000000_7FFA7610  00000000/00000000_00000000
AR/GR 2-3     01FF000B/00000000_7FFA7500  00000000/00000000_7FFA7500
AR/GR 4-5     00000000/00000000_00000110  00000000/00000000_00000002
AR/GR 6-7     00000000/00000000_7FFA7500  00000000/00000000_00000110
AR/GR 8-9     01FF000B/00000000_7F6A89D8  00000000/00000000_00000000
AR/GR 10-11   00000000/00000008_00300000  00000002/00000008_00300000
AR/GR 12-13   00000002/00000000_0B0BBFC0  00000000/00000000_7F6A89F8
AR/GR 14-15   01FF000B/00000000_01701BF8  00000401/00000000_00000000

Home ASID: 0340    Primary ASID: 0365    Secondary ASID: 0340
```

Interrupt Code = PIC10:
PSW points AT failing instr

Instruction Length = 6:
Failing inst is D503B000C02B

IP OPCODE D503B000C02B
CLC instruction

Instruction base registers are
Reg11 and **Reg12**

TEA = 00000008_00300801
Note additional line identifying access register

NOTES: The CLC instruction suffered a PIC10 while trying to move data from above the bar to below the bar in the home address space, ASID 340. The high virtual storage reference caused the PIC10. Could the ALET be incorrect such that the code really meant to access storage in another space? Could the AMODE be incorrect such that the program really meant to access storage below the bar? Could the AMODE be correct but the high order word of the address in R11 should not be “dirty”?

Here is an example of an ABEND0C4 PIC10 that occurs under a program running in AR mode. Here we see that a CLC instruction failed and TEA matches base register 11. Additionally note that the ALET in Access Register 11 is 00000002 which means that the address in general purpose register 11 is to be mapped to the home address space ASID 340. When trying to understand the reason for the program check, the debugger must consider whether this was the intended environment.

Diagnosing program checks: Summary

- Need to gather program check-related data including:
  - Program Interrupt Code
  - Instruction Length
  - Failing instruction
  - PSW & Registers
  - TEA (for Translation Exceptions)
  - Storage key (for Protection Exceptions)
- Need to be sensitive to execution environment
  - Addressing mode (AMODE)
  - Cross memory mode
  - Access register mode
- "Gotchas"
  - Instructions with displacements/ lengths that cause data to be moved across page boundaries
  - Instructions with index registers
  - Failure on instruction fetch rather than data access

Diagnosing program checks is an important skill. Data gathering is straightforward. Analysis is not too difficult so long as you remember to pay attention to the details, including the environment (amode, cross memory mode, AR mode) and the base/index/displacement/length in the failing instruction.