Health Checker z/OS 1.12 and Red Paper 4590

Session 7520

Peter Relson
IBM Poughkeepsie
relson@us.ibm.com
4 August 2010

Trademarks

Abstract

The presentation will cover the new functions, facilities, and checks for the IBM Health Checker for z/OS that will be available in z/OS 1.12. In addition, to take advantage of the brand new Health Checker RedPaper 4590 which is a great addition to the IBM Health Checker for z/OS User's Guide, the presentation will discuss various aspects of that RedPaper, focusing on things that have not previously made their way into Share presentations, such as persistent data. The presentation will also go into some of the RedPaper's great examples of neat (and useful) things you can do yourself to exploit the capabilities of the IBM infrastructure, including use of Metal C and non-SYSREXX REXX, and message management through MPF exits.

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

Metal C

- Metal C is supported as an additional check writing language
- We all know that C is familiar to new developers and developers from many different platforms
- Metal C can be used to write local (authorized) and remote (authorized or not authorized) checks
- It does not require any unique Health Checker syntax (as it uses the Health Checker assembler macros). This is contrasted with System REXX which, for example, has a number of unique functions provided to help it work

Why Metal C?

- METAL C does not require a language environment (LE)
- **METAL C allows you to embed assembler statements and in particular assembler (macro) service invocations**
- METAL C supports all the core functions of “regular” C
  - Although standard library functions are not generally supported, some can be replaced by corresponding assembler services if needed

Metal C Support

- No callable services are provided, but rather z/OS provides
  - mappings for structures and constants
- The coder is expected to use the mappings to set up the data that is needed and use the Metal C support for parameterized invocations of assembler macros to access IBM Health Checker for z/OS functions

Metal C Header Files

- HZSHPQE: analog of assembler macro HZSPQE
- HZSHDPQE: analog of HZSDPQE
- HZSHMGB: analog of HZSMGB
- HZSHQUAA: analog of HZSQUAA
- HZSHCPAR: analog of HZSZCPAR
- HZSHENF: analog of HZSZENF
- HZSHHCKL: analog of HZSZHCKL
- HZSHCONS: analog of HZSZCONS
- Provided as C header files in `SYS1.SIEAHDR.H`
  - For example, **`SYS1.SIEAHDR.H(HZSHPQE)`**
  - Can be included in METAL C as “normal” include:
    - **#include <hzshpqe.h>**
- Header HZSH can be used to include all of the above

Metal C samples

- Code samples are shipped in **/usr/lpp/bcp/samples**
  - To let you quickly try out METAL C for health checks
  - Have working examples of Health Checker service invocations via the _asm syntax
- hzscadd.c – HZSADDCHECK exit routine, use of HZSADDCK
- hzscchkp.c – sample check routine, use of HZSFMSG, HZSPREAD, HZSPWRIT
- hzscchkr.c – sample check routine, use of HZSFMSG
- hzscrchc.c – sample remote check routine, use of HZSCPARS
- hzscrchk.c – sample remote check routine
- hzssmake.mk is a “Makefile” to build the above using the Unix utility “make”

Metal C examples (HZSADDCK)

- HZSADDCK (in HZSCADD)
- Struct with data

```
/* This structure is used to store parameter information
   for invoking HZSADDCK */
typedef struct checkInfo_s {
    int   entry;       /* unique val for a check */
    char  name[32];    /* check name             */
    char  date[8];     /* YYYYMMDD               */
    int   reasonLen;   /* 1 - 126                */
    char *reason;      /* Up to 126 char string  */
    int   parmsLen;    /* 1 - 256                */
    char *parms;       /* Up to 256 char string  */
} checkInfo;
```

- Fill in the struct fields with appropriate data

Metal C examples (HZSADDCK) cont

- HZSADDCK (in HZSCADD)

```
__asm(" HZSADDCK ACTIVE,"
"SEVERITY=LOW,"
     "INTERVAL=ONETIME,"
     "USS=NO,"
     "ENTRYCODE=0(%3),"  /* Based on checkInfo struct */
     "CHECKNAME=4(%3),"
     "DATE=36(%3),"
     "REASONLEN=44(%3),"
     "REASON=(%4),"
     "PARMS=(%5),"
     "PARMSLEN=52(%3),"
     "CHECKOWNER=ckOwner,"
     “EXITRTN=exitRtn,"
     "CHECKROUTINE=checkRtn,"
     "MSGTBL=msgTbl,"
     "RETCODE=(%0),"
     "RSNCODE=(%1),"
     "MF=(E,(%2),COMPLETE)"
     : "=r"(rc), "=r"(rsn)
     : “r”(&addCk), “r"(&checkInfo), "r"(checkInfo.reason),
       "r"(checkInfo.parms));
```

Metal C __ASM Substitution Rules

- **Between the first and second colons are "output" things**
  - "=r" means to select register n, substitute with "n" and store from register n into the variable "after"
  - "=m" means to select register n, substitute with "0(n)" and set the variable's address into register n "before"
- **After the second colon are "input" things**
  - "r" means to select register n, substitute with "n" and to load into register n from the variable "before" or set the variable's address into register n "before"
  - "m" means to select register n, substitute with "0(n)" and to set the variable's address into register n "before"
- **The behavior above is what is important. Use what behaves the way you need it to. But don't lie to the compiler**
- See the Metal C book section 1.2.5 "Inserting HLASM instructions into the generated source code"

Metal C examples (HZSADDCK) cont

- HZSADDCK (in HZSCRCHC)

```
__asm(" HZSADDCK "
        “CHECKOWNER==CL16'IBMSAMPLE',"
        "CHECKNAME==CL32'HZS_SAMPLE_REMOTE_MC_HZSCPARS',"
        "ACTIVE,"
        "SEVERITY=LOW,"
        "REMOTE=YES,"
        "USS=NO,"
        "HANDLE=%4,"
        "PETOKEN=%3,"
        "INTERVAL=ONETIME,"
        "VERBOSE=NO,"
        "DATE==CL8'20090212',"
        "REASON==CL41'Sample Metal C health check with HZSCPARS',"
        "REASONLEN==A(41),"
        "PARMS==CL38'PARM1(1,999),PARM2(100),PARM3(CHOICE1)',"
        "PARMSLEN==A(38),"
        "RETCODE=%2,"
        "RSNCODE=%1,"
        "MF=(E,(%0),COMPLETE)"
       : : "r"(&PListAddCk), "m"(rsncode), "m"(retcode),
           “m”(PEToken), "m"(checkHandle) );
```

Metal C examples (HZSFMSG)

- HZSFMSG

```
__asm(" HZSFMSG REQUEST=CHECKMSG,“
     "MGB=(%3),“
     "RETCODE=(%0),“
     "RSNCODE=(%1),“
     "MF=(E,(%2))“
     : "=r"(hzsfmsg_rc)
     , "=r"(hzsfmsg_rsn)
     : “r”(&fMsgParmLst)
     , "r"(&MGB));
```

Metal C examples (HZSPREAD)

- HZSPREAD (in HZSCCHKP)

```
__asm(" HZSPREAD CHECKOWNER=ckOwner,"
     "CHECKNAME=ckName,"
     "IPL=PRIOR,"
     "INSTANCE=MOSTRECENT,"
     "STARTBYTE=0,"
     "BUFFER=%0,"
     "DATALEN=%4,"
     "BYTESAVAIL=%1,"
     "RETCODE=(%2),"
     "RSNCODE=(%3),"
     MF=(E,(%5),COMPLETE)"
     : "=m"(PReadBuffer)
     , "=m"(PReadBytes)
     , "=r"(hzspread_rc)
     , "=r"(hzspread_rsn)
     : "m"(persistentDataBytes)
     , “r”(&lRead));
```

Metal C examples (HZSPWRIT)

- HZSPWRIT (in HZSCCHKP)

```
__asm(" HZSPWRIT BUFFER=&2,“
     "DATALEN=%4,“
     "RETCODE=%0,“
     "RSNCODE=%1,“
     "MF=(E,(%3),COMPLETE)“
     : "=m"(hzspwrit_rc)
     , "=m"(hzspwrit_rsn)
     : “m”(PWriteBuffer)
     , "r"(&lWrit)
     , "m"(persistentDataBytes));
```

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

Message Table

- Ever since the Health Checker framework was introduced, a health check has consisted of two main parts:
    - the check routine (code) and
    - the message table (msg data)
- The process of defining the content of the message table and creating the message table can be unwieldy and feels unnecessary for REXX check writers since REXX has such good string-manipulation functions.

Why even have a Message Table?

- Makes it easier to provide message translations for different languages
- The “rigid” structure of the message table input allows for
  - Consistent appearance of check messages across health checks
  - Enforcement of completeness of check message contents
  - Programmatic approach to generation of message documentation
- IBM supplied health checks will continue to use message tables.
- Nevertheless ....

No Message Table for REXX checks

- REXX users wanted to be able to create their message text using standard REXX functions. They could do so but would still need to create an artificial message table simply to hold the (complete) text that they set up
- In z/OS V1R12 any check message text may be provided directly by the check routine, therefore removing the requirement to provide a separate message table.

No Message Table: Adding a check

- The ADD | ADDREPLACE HZSPRMxx parmlib statement still requires a **message table** parameter, but a special value ***NONE** can now be specified.
- For example:

```
ADDREP CHECK(IBMHZS,HZS_DIRECTMSG_REXX)
       EXEC(UTH1472R)
       REXXHLQ(UTHUSER)
       REXXTSO(NO)
       REXXIN(NO)
       MSGTBL(*NONE)
       USS(NO)
       SEVERITY(HIGH)
       INTERVAL(ONETIME)
       ACTIVE
       DATE(20091003)
       REASON('DIRECTMSG REXX SAMPLE')
```

No Message Table: REXX check syntax

- Main target are REXX checks due to REXX's powerful string manipulation capabilities
- The existing REXX function **HZSLFMSG** now supports
  - a new **HZSLFMSG_REQUEST** type of “DIRECTMSG”
  - with new **HZSLFMSG_REASON**s “CHECKEXCEPTION”, “CHECKINFO”, and “CHECKREPORT”
- For example:

```
HZSLFMSG_REQUEST        = “DIRECTMSG”
HZSLFMSG_REASON         = “CHECKINFO”
HZSLFMSG_DIRECTMSG_ID   = “CNZH0001I”
HZSLFMSG_DIRECTMSG_TEXT = “Sample msg…”
CALL HZSLFMSG()
```

No Message Table: Assembler checks

- Besides REXX function HZSLFMSG, the assembler **macro HZSFMSG** now also allows for REQUEST=DIRECTMSG…
- Small difference: Text parameters need a corresponding text length parameter
- For example:

```
HZSFMSG REQUEST=DIRECTMSG,
      REASON=CHECKREPORT,
      TEXT=reportline,
      TEXTLEN==AL2(L'reportline),
      MF=(E,HZSFMSG_List)
```

- We suggest that a real message table is the better way to go for non-REXX checks

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

SuperUser

- In order to successfully run health checks which use z/OS Unix, Health Checker requires a user profile with
    - OMVS segment and
    - z/OS Unix super user authority
  - ➢ Previously, super user authority was required to be given via a user profile with uid(0)
  - ➢ Uid(0) user profiles are closely watched by auditors and might require extra explaining and might even prevent users from exploiting Health Checker

SuperUser (cont)

- In z/OS V1R12 you can associate the Health Checker address space with a user profile which has
  - READ access to the BPX.SUPERUSER resource in the FACILITY class, and
  - A non-zero uid

- **Result**
  - The Health Checker associated user profile does not appear on any uid(0) audit reports anymore.
  - **BUT note that the underlying authorization requirement still exists and is persistent for the life of hzsproc.**

SuperUser (cont)

- The IBM Health Checker for z/OS User's Guide has details and example statements for both associating the Health Checker address space
  - With a uid(0) user profile
    - **ADDUSER *hcsuperid* OMVS(UID(0)…**
    - **RDEFINE STARTED HZSPROC.* STDATA(USER(*hcsuperid*)…**
  - or
  - With a BPX.SUPERUSER user profile
    - **ADDUSER *hcsuperid* OMVS(UID(*non-zero*)…**
    - **PERMIT BPX.SUPERUSER CLASS(FACILITY) ID(*hcsuperid*) ACCESS(READ)**
    - **RDEFINE STARTED HZSPROC.* STDATA(USER(*hcsuperid*)…**

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

SDSF Health Check History

- The IBM Health Checker provides an option for saving Health Checks to a logstream for historical purposes.
- SDSF can now let the user view either the current status of health checks via the SDSF Health Checker display or the history from the logstream.

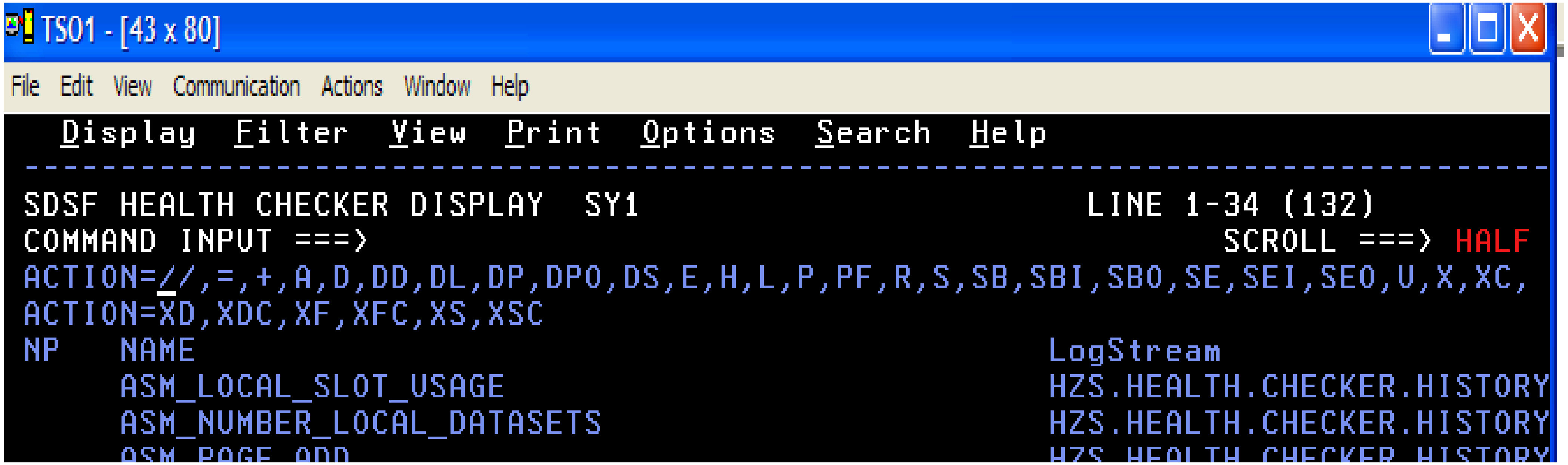
SDSF Health Check History (cont)

You can:

- View the frequency and results of previous health check runs
- Compare historical results of health checks to evaluate the results of updating component parameters
- Browse, save or print specific health check runs

SDSF Health Check History (cont)

- In support of the new SDSF Health Checker History display there is a new column on the SDSF Health Checker display titled LogStream. This column shows the current logstream to which the IBM Health Checker for z/OS is connected, from which history data will be gathered.



```
  Display  Filter  View  Print  Options  Search  Help
 ---------------------------------------------------------------------------
 SDSF HEALTH CHECKER DISPLAY  SY1                      LINE 1-34 (132)
 COMMAND INPUT ===>                                           SCROLL ===> HALF
 ACTION=//,=,+,A,D,DD,DL,DP,DPO,DS,E,H,L,P,PF,R,S,SB,SBI,SBO,SE,SEI,SEO,U,X,XC,
 ACTION=XD,XDC,XF,XFC,XS,XSC
 NP   NAME                                             LogStream
      ASM_LOCAL_SLOT_USAGE                             HZS.HEALTH.CHECKER.HISTORY
      ASM_NUMBER_LOCAL_DATASETS                        HZS.HEALTH.CHECKER.HISTORY
      ASM_PAGE_ADD                                     HZS.HEALTH.CHECKER.HISTORY
```

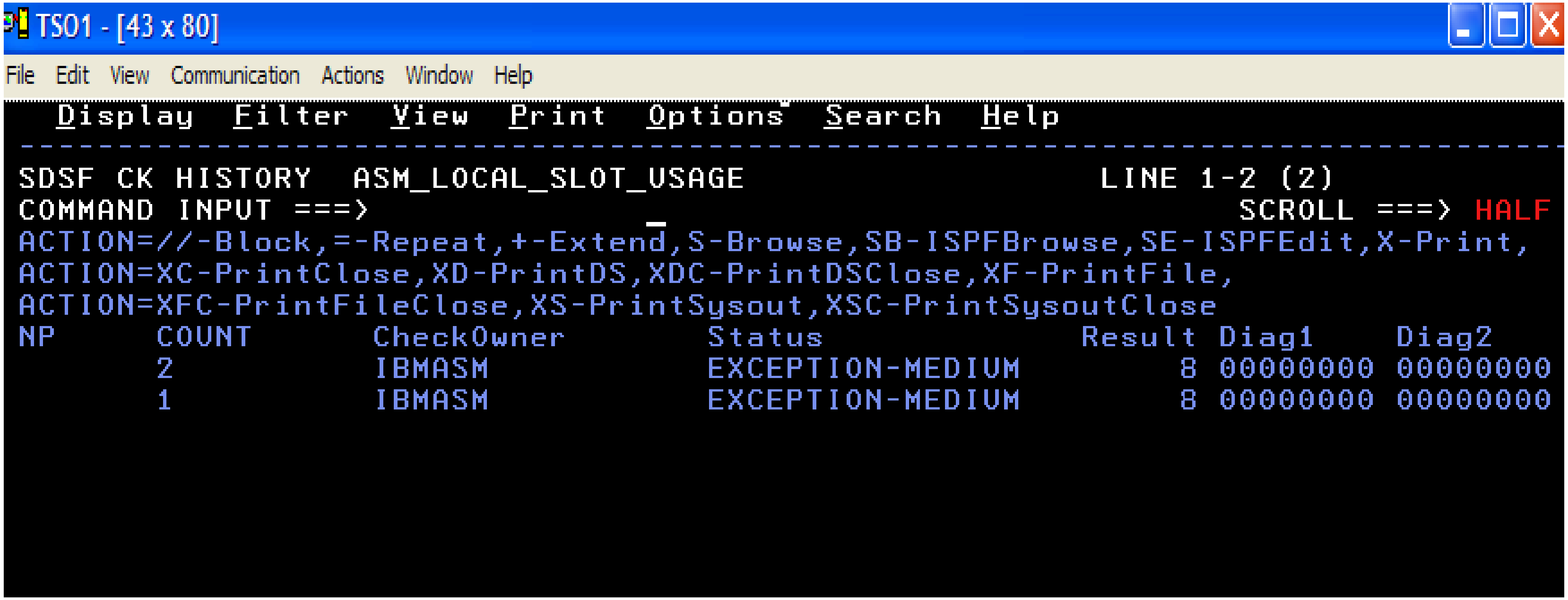
SDSF Health Check History (cont)

The support is invoked by:

- The new L (ListHistory) action on the SDSF Health Checker display
- This action is valid for checks that have had their output written to the logstream

SDSF Health Check History (cont)

The L action presents the user with the SDSF Health Checker History display:



```
  Display  Filter  View  Print  Options  Search  Help
-------------------------------------------------------------------------------
SDSF CK HISTORY  ASM_LOCAL_SLOT_USAGE                       LINE 1-2 (2)
COMMAND INPUT ===>                                                 SCROLL ===> HALF
ACTION=//-Block,=-Repeat,+-Extend,S-Browse,SB-ISPFBrowse,SE-ISPFEdit,X-Print,
ACTION=XC-PrintClose,XD-PrintDS,XDC-PrintDSClose,XF-PrintFile,
ACTION=XFC-PrintFileClose,XS-PrintSysout,XSC-PrintSysoutClose
NP      COUNT       CheckOwner        Status                 Result Diag1    Diag2
        2           IBMASM            EXCEPTION-MEDIUM            8 00000000 00000000
        1           IBMASM            EXCEPTION-MEDIUM            8 00000000 00000000
```

SDSF Health Check History (cont)

- From the Health Checker History display the user can:
  - Browse a specific run of a check
  - Print a specific run of a check
- The information provided is for each run of a check and includes:
  - Check owner
  - Status
  - Result
  - Diagnostic data
  - Start and stop time
  - System and sysplex name
  - Check name which is also seen on the panel title line

SDSF Health Check History (cont)

- By default, SDSF will collect the last 10 iterations of a check. The user can override this default using the new SET CKLIM command. The minimum number of checks can be 1 and the maximum is 999,999.

- You can also override the SDSF default of 10 iterations using the new Panel.CK.DefaultCKLim custom property. Again, the user can override this default via the SET CKLIM command

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

Persistent Data

- Services available to checks to save and retrieve data between IPLs
- Persistent data is saved in HZSPDATA DD (data set).
  - HZSPDATA may have to be resized as persistent data users increase over time.
- HZSPWRIT - Write service for persistent data.
- HZSPREAD - Read service for persistent data.

HZSPDATA data set

- The data set captures
  - Data from the first request (by each exploiting check) for this IPL
  - Data from the most recent request (by each exploiting check) for this IPL
- The data set saves
  - Data from the previous IPL by each exploiting check (first and most recent requests)
- The system writes the persistent data to the data set periodically, but not until one hour has elapsed since hzsproc has been started (if you had made some mistake, you have that hour to terminate hzsproc without having changed the data set). Once hzsproc has been up at least an hour, the latest data will be written to the data set if hzsproc is stopped.

HZSPDATA data set (cont)

- Specified in hzsproc via HZSPDATA DD
- Fixed block, record length 4096 data set
- If there is a problem, the system indicates how big it needs to be and will prompt for a new (empty) data set
  - Messages HZS0012E, HZS0013A
- Do not edit the data set
- At HC startup, the data set is read, and all its data is buffered in storage (data for the current IPL is initially empty, data for the previous IPL is buffered). Suppose the data set had had data from IPL's 1 and 2 and now this is IPL 3. The data from IPL 1 will be discarded. The data from IPL 2 becomes “previous IPL's data” and the forthcoming data from IPL 3 is the current IPL's data

Write to HZSPDATA

- **HZSPWRIT service**
  - Used by a check routine
  - The check routine may write one time or multiple times. All the data written in one check iteration is concatenated together and maintained in virtual storage. Thus it makes no difference to the system if the check routine uses HZSPWRIT once to write all the data or multiple times to write each piece.

Write to HZSPDATA

- **HZSPWRIT service (cont)**
  - The check routine identifies
    - Its handle (if a remote check). It is assumed to be the currently running check if a local check.
    - The data to write and how much data
  - The system buffers the data in virtual storage
  - Upon completion of the check iteration, the data is available to be read
  - The data is written to the data set periodically
  - There is no way or need to ask that the data be written “now” to the data set

Write to HZSPDATA (example)

- **HZSPWRIT service (cont)**
  - HZSPWRIT BUFFER=buffer,
    DATALEN=datalen,
    RETCODE=rc,RSNCODE=rsn,
    MF=(E,hzspwrit_listform)
  - Buffer is the buffer from which data is to be written
  - The amount of data to be written.

Read from HZSPDATA

- **HZSPREAD service**
  - Used by a check routine (usually the check routine that wrote the data)
  - The check routine that wrote the data is responsible for its format. The system has no idea. If the check routine has not externalized that format, you should not make any assumptions about that format.
  - Security checking is done (always if unauthorized caller, on request if authorized) against XFACILIT class entity
    - HZS.sysname.checkowner.PDATA
    - HZS.sysname.checkowner.checkname.PDATA

Read from HZSPDATA

- **HZSPREAD service (cont)**
  - The issuer identifies
    - The requested check owner/name (HZSPREAD need not be issued from the check that wrote the data, although that is the typical usage)
    - Its handle (if a remote check). It is assumed to be the currently running check if a local check.
    - A target buffer into which to write the data
    - The “start byte” of data to be read (for example, it might read the first 4096 bytes, and then ask to start the next time with the 4097th byte (a value of 4096).
  - The data returned is from the virtual storage buffer. The data set is not accessed

*

Read from HZSPDATA

- **HZSPREAD service (cont)**
  - The issuer identifies
    - Which copy of the data to be read
      - Prior IPL's data, first iteration
      - Prior IPL's data, most recent (last) iteration
      - This IPL's data, first iteration
      - This IPL's data, most recent
    - What security checking to do

Read from HZSPDATA (example)

- **HZSPREAD service (cont)**
  - ➢ HZSPREAD CHECKOWNER=owner,

```
CHECKNAME=name
IPL=CURRENT,
INSTANCE=MOSTRECENT,
BUFFER=ibuffer,DATALEN=datalen,
STARTBYTE={startbyte | FIRST_BYTE},
BYTESAVAIL=bytesavail,
RETCODE=rc,RSNCODE=rsn,
MF=(E,hzspread_listform)
```

Read from HZSPDATA (example)

- **HZSPEAD service (cont)**
  - Buffer is the buffer into which the data is to be read
  - Datalen is the number of bytes to be read
  - Startbyte identifies the starting byte to be read (on first call, usually 0)
  - Bytesavail is output by the system indicating how many bytes were available to be read. This can be used to gauge how big a buffer to allocate for a subsequent call

HZSPDATA protocol

- A typical protocol for a check iteration might be
  - HZSPREAD to request the most recently written data for this IPL (i.e., “what did I write the last time?”)
  - If that does not exist (then I have never run for this IPL) request the most recent data from the previous IPL
  - For both, start reading at byte 0. Provide a buffer big enough to hold the data, possibly re-allocated after learning how big an area is needed
  - If no copy is found then I must assume I have no prior data to worry about
  - Manipulate the data, create new data, whatever
  - HZSPWRIT (one or more invocations as convenient) to write the data

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

IOSSPOF service

- Service to detect single point of failure
- Available as of z/OS R10
- Used by many checks (and can be used by you)
- But most important is its example of showing how you can provide a service that writes to the caller's check's message buffer on behalf of that check

IOSSPOF service (cont)

- HealthChecker-wise, IOSSPOF takes as input
  - Does the caller want HC messages produced or not (HCMSG=NO | YES)
  - When "YES", is the caller a remote check (as opposed to a local check) and if remote what is the check's handle (HANDLE=xhandle)
  - These enable the service to know what parameters to specify when it invokes HZSFMSG to write a HC message.
    - When HANDLE is not provided, uses REMOTE=NO
    - When HANDLE is provided, uses REMOTE=YES, HANDLE=xhandle

IOSSPOF (cont)

- Created its own message table using normal HC functionality, but this message table is not managed in any way by HC (in this case, it happens to be packaged with the IOSSPOF service load module)
- On its HZSFMSG invocation, the IOSSPOF service specifies MSGTABLE=iosspof_msgtable
  along with REMOTE=YES,HANDLE=xhandle or REMOTE=NO according to the IOSSPOF user's parameters

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

Other RedPaper goodies

- Use of MPF to run a check when a particular message is issued
- Use of MPF to get notification upon important messages
- Use of non-SYSREXX REXX for check development

Message-driven check running

- Create your check routine (perhaps via SYSREXX)
- Add your check via HZSPRMxx ADDREP statement identifying the check routine and (if it exists) message table
- Create an MPF exit “your_exit”
- Define in MPFLSTxx the message you want to watch for and the exit to get control when that happens (“your exit”) ABCnnnI,SUP(NO),USEREXIT(your_exit)
- In your exit, issue HZSCHECK REQUEST=RUN for the particular check that you want to run

Important Messages Check(s)

- The situation: A lot of customers like the exception notification capabilities of HC. They want things not even necessarily reported by HC to be surface-able by this mechanism. How about system messages?
- Suppose you have a list of messages for which you would like to get HC-based notification and then be able to browse the message buffer for a check to look at the data
- This is a an example covered in the Red Paper so for the full details, consult that document
- IBM has been asked to create something like this.
- In the mean time, this could get you started

Important Messages Check(s) (Cont)

- Here are the main pieces involved
  - Identify to the system which messages you want tracked
  - Add a check for each message (yes, one check per message)
  - Create an MPF exit to capture the message text when the message is issued,
  - Create a check routine to write the saved message text to the check's message buffer (coupled with a message table)

Important Messages Check(s): Identify

- MPF processing is the key. For each message of interest, set up an entry in MPFLSTxx that ties the message to an exit routine that you will provide
- Within MPFLSTxx, for example,
  - IEE611I,SUP(NO),AUTO(Y),USEREXIT(yourexit)
- "Yourexit" will be given control when IEE611I is issued
- You will also need to have your CONSOLxx parmlib member point to your MPFLSTxx via the MPF keyword (initially you may activate via the SET MPF command)

Important Messages Check(s): Add Check

- Within HZSPRMxx, activated usually by
  F hzsproc,ADD PARMLIB=xx. Statements such as
- ADDREP CHECK(owner,MESSAGE_ABCnnnI)
  MSGTBL(the_message_table)
  PARM(ALL)
  VERBOSE(NO)
  SEVERITY(MEDIUM)
  INTERVAL(hh:mm)
  DATE(20100804)
  REASON('alert for message ABCnnnI')
- One such statement for every message being tracked

*

Important Messages Check(s): MPF exit

- Routine #C1HZS00 within the RedPaper chapter 9 package
- This routine allocates ECSA in which to hold the messages as they are issued, with one 64K area per message. It could use >2G common storage instead.
- A system-level name/token is used to locate the data for sharing between this routine and the check routine
- After setting up the data, the MPF exit can request that the check be run via the HZSCHECK macro, so that the notification will result.

Important Messages Check(s): Check Routine

- Routine #C3HZSCK within the RedPaper chapter 9 package
- This routine retrieves the information saved by the MPF exit and writes it to the check's message buffer
- It implements a protocol whereby to “start over” for a given message, you must refresh the check. That is not an ideal protocol, but it does work.

Checks in non-SYSREXX REXX

- When developing checks that will ultimately be SYSREXX, it can be nice to be in a non-authorized environment
- REXX is just another programming language.
- You need eventual access to the assembler services that underly the functions with which any remote check must interact
- The same can be said for non-Metal C
- The goal is to be able to have your REXX exec, once debugged, moved with minimal changes to a SYSREXX library

Checks in non-SYSREXX REXX (cont)

- SYSREXX checks get input in REXX variables and communicate with HC services (such as the HZSCHECK macro through HZSLSTRT / HZSLSTOP and the HZSFMSG macro through HZSLFMSG) via REXX variables
- You need to provide a front-end that sets up the REXX variables that the check exec will receive
- You need to provide stubs that the check exec can invoke that accomplish what will happen when the SYSREXX exec invokes HZSLxxxx.

Checks in non-SYSREXX REXX (cont)

- Stubs provided with RedPaper chapter 8 package
  - XZSLADCK – check initialization (load message table, allocate pause element, add the check)
  - XZSLSTRT – HZSCHECK REQUEST=OPSTART
  - XZSLSTOP – HZSCHECK REQUEST=OPSTOP
  - XZSLFMSG – HZSFMSG
  - XZSLCLNP - release the pause element, delete the message table

Agenda

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**
- **Summary**

New checks in R12

- Communications Server
  - (IBMCS,CSTCP_IPMAXRT4_tcpipstackname)
  - (IBMCS,CSTCP_IPMAXRT6_tcpipstackname)
- DFSMS SMS
  - (IBMSMS,SMS_CDS_SEPARATE_VOLUMES)
  - (IBMSMS,SMS_CDS_REUSE_OPTION)
- DFS SMB
  - (IBMSMB,SMB_NO_ZFS_SYSPLEX_AWARE)
  - (IBMSMB,ZOSMIGREC_SMB_RPC)

New checks in R12 (cont)

- XCF/XES
  - (IBMXCF,XCF_SFM_CFSTRHANGTIME)
  - (IBMXCF,XCF_CF_Processors)
  - (IBMXCF,XCF_CF_Memory_Utilization)
  - (IBMXCF,XCF_CDS_MAXSYSTEM)
  - (IBMXCF,XCF_CFRM_MSGBASED)
  - (IBMXCF,XCF_CF_STR_POLICYSIZE)
- IOS
  - (IBMIOS,IOS_CAPTUCB_PROTECT)
  - (IBMIOS,IOS_STORAGE_IOSBLKS)
  - (IBMIOS,IOS_MIDAW)

New checks in R12 (cont)

- z/OS UNIX
  - (IBMUSS,USS_HFS_DETECTED)
- Infoprint
  - (IBMINFOPRINT,INFOPRINT_INVENTORY_CHECK)
  - (IBMINFOPRINT,ZOSMIGV1R12_INFOPRINT_INVSIZE)
- Supervisor
  - (IBMSUP,SUP_LCCA_ABOVE_16M)
- Reconfiguration
  - (IBMRCF,RCF_PCCA_ABOVE_16M)

Check Details – Communications Server

(IBMCS,CSTCP_IPMAXRT{4|6}_*tcpipstackname*)

- Checks whether the total number of IPv{4|6} indirect static and dynamic routes in the TCP/IP stack's routing table exceeds a maximum threshold. The *tcpipstackname* suffix is the job name of the TCP/IP stack to which this check applies.
- A high number of routes added by OMPROUTE and the TCP/IP stack can potentially result in high CPU consumption from routing changes. A large routing table is considered to be inefficient.
- Parameter ‘IPMAXRT{4|6}(*n*)’ specifies an integer value indicating the maximum threshold value for the number of IPv{4|6} indirect static and dynamic routes that the TCP/IP stack can add to its routing table before issuing a warning. Default is *n*=2000
- Interval WEEKLY, Severity LOW

Check Details – DFSMS SMS

(IBMSMS,SMS_CDS_SEPARATE_VOLUMES)

- Verifies that the active control data set (ACDS) and communications data set (COMMDS) are not residing on same volume.
- To ease recovery in case of failure. Also, you should allocate a spare ACDS on a different volume. The control data set (ACDS or COMMDS) must reside on a volume that is not reserved by other systems for a long period of time because the control data set (ACDS or COMMDS) must be available for SMS processing to continue.
- No parameters
- Interval ONETIME, Severity MEDIUM

Check Details – DFSMS SMS

(IBMSMS,SMS_CDS_REUSE_OPTION)

- Verifies that the active control data set (ACDS) and communications data set (COMMDS) are defined with the REUSE option.
- Helps to avoid running into space problems (SMS reason code 6068) as result of subsequent ACDS or COMMDS updates, or IMPORT/EXPORT functions.
- No parameters
- Interval ONETIME, Severity MEDIUM

Check Details – DFSMS SMB

(IBMSMB,SMB_NO_ZFS_SYSPLEX_AWARE)

- Determines if the DFS/SMB File Server is running in a sysplex and if so, determines if any member of the sysplex is running zFS sysplex aware
- In a sysplex environment, exportation of a zFS file system and subsequent sharing by the DFS/SMB server can only take place on the system that owns the file system and is not running zFS sysplex aware. Beginning with z/OS V1R11, the SMB server cannot export zFS read/write file systems when zFS is running sysplex-aware on either:
  - ➢ The same system where the SMB server is running
  - ➢ The system that owns the zFS file system
- No parameters
- Interval ONETIME, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_SFM_CFSTRHANGTIME)

- Monitors the CFSTRHANGTIME value in the Sysplex failure management (SFM) policy to make sure it is specified and it matches the check parameter CFSTRHANGTIME value. CFSTRHANGTIME specifies the amount of time before the system takes automatic action to relieve hangs caused when a connector fails to respond to structure-related events in a timely manner.
- Installation should specify SFM policy parameter CFSTRHANGTIME so that such hangs can be relieved automatically.
- Parameter 'CFSTRHANGTIME(NO | *seconds*)‘ (default is seconds=300) specifies the wait time interval, in seconds, or NO, if no automatic relieve is requested
- Interval 4:00, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_CF_Processors)

- Provides a warning when a coupling facility processor configuration is not consistent with IBM recommendations and may result in degraded response time and throughput. Based on the coupling facility architected function level (CFLEVEL).
- Parameter '*EXCLUDE(CFname, CFName,…)*' allows to specify a list of CFNames that the check should not consider in its verification processing. Processor configurations for excluded CFs will still be reported on, just not factored into the overall check status. Default is 'EXCLUDE()' (no exclusions).
- Interval 4:00, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_CF_Memory_Utilization)

- Raises an exception when a coupling facility reaches a given memory utilization percentage threshold.
- Prevents a CF from becoming over-full and thereby allows the CF to allocate new structures, expand structures, sustain a viable failover environment and participate in structure rebuild and reallocation processing when needed.
- Parameter ‘MAXUTILIZATION(*n*[%])’ indicates the threshold percentage that the Coupling Facility memory utilization should not exceed. Default is 60%.(
- Interval 1:00, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_CDS_MAXSYSTEM)

- Warns when a function CDS (any CDS other than the Sysplex CDS) is formatted with a MAXSYSTEM value that is less than the MAXSYSTEM value associated with the primary Sysplex CDS.
- If a function CDS has a smaller MAXSYSTEM value, then a system joining the Sysplex with a higher slot number will not be able to use the function provided by that function CDS.
- No parameters
- Interval ONETIME, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_CFRM_MSGBASED)

- Checks to see if CFRM is enabled to use a given event processing protocol when a structure is defined in the CFRM active policy in a multisystem-capable sysplex.
- CFRM message-based event management can improve performance, availability, and scalability for users of CF structures.
- Parameter '{MSGBASED|POLBASED}' specifies that the check should issue an exception if the sysplex is not using {message-based | policy-based} CFRM processing. Default is MSGBASED.
- Interval 4:00, Severity MEDIUM

Check Details – Cross system coupling facility

(IBMXCF,XCF_CF_STR_POLICYSIZE)

- Check that structures in the CFRM active policy do not differ too much for values INITSIZE and SIZE. INITSIZE should indicate an initial structure size of at least half the maximum structure size (SIZE). INITSIZE should not be less than the maximum structure size when altering of the structure size is not supported.
- Specifying different INITSIZE and SIZE values provides flexibility to dynamically expand the size of a structure for workload changes, but too large a difference between INITSIZE and SIZE may waste coupling facility space or prevent structure allocation.
- No parameters
- Interval 4:00, Severity MEDIUM

Check details – Input/Output Supervisor

(IBMIOS,IOS_CAPTUCB_PROTECT)

- Verifies that captured UCB (Unit Control Block) protection is active on the system.
- UCBs are control blocks in storage that define the characteristics of devices. Legacy software may require a subset of these to reside in the first 16 megabytes of storage. There is a service to capture the UCB and temporarily put it in this 24-bit addressable area. Captured UCB Protection places the UCBs in write protected storage, which is recommended by IBM.
- Parameter ‘PROTECT({YES|NO})’ indicates the expected state of captured UCB protection. Default is YES – the captured UCB protection state should be enabled.
- Interval ONETIME, Severity MEDIUM

Check details – Input/Output Supervisor

(IBMIOS,IOS_STORAGE_IOSBLKS)

- Verifies that control blocks used in IOS can reside in 31-bit addressable storage.
- Control blocks used to initiate I/O were obtained in storage addressable in the first 16 megabytes of storage so that 24-bit AMODE legacy software could perform scans on them. However, forcing all I/O control blocks below the 16 megabyte line creates a constraint on the amount of storage below the line. IBM recommends that the control blocks be allowed to be in 31-bit addressable storage.
- Parameter 'IOSBLKS({24|31})', indicates the expected actual state of 31-bit-allowed IOS blocks. Default is 31.
- Interval ONETIME, Severity LOW

Check details – Input/Output Supervisor

(IBMIOS,IOS_MIDAW)

- Verifies that the modified indirect addressing word (MIDAW) facility is enabled.
- MIDAWs are a more efficient way to issue I/O commands and allow better use of I/O bandwidth.
- Parameter ‘MIDAW({YES|NO})’ indicates the expected state of MIDAW. Default is YES – MIDAW should be enabled.
- Interval ONETIME, Severity LOW

Check details – z/OS Unix System Services

(IBMUSS,USS_HFS_DETECTED)

- Issues an exception if any mounted HFS file systems are found and owned by the system running the health check.
- HFS file systems are no longer the strategic file system. All HFS file systems should be migrated to zFS.
- Parameter ‘RUN_ON_MOUNT={YES|NO}’ indicates whether or not the check should run after the successful mount of an HFS file system. RUN_ON_MOUNT=YES is the default.
- Parameter ‘HFS_LIST=(*FSname, FSName,…*)’ indicates a list of HFS file systems (-data sets) that you wish this check to ignore.
- Interval 24:00, Severity LOW

Check details – Infoprint Server

(IBMINFOPRINT,INFOPRINT_INVENTORY_CHECK)

- Warn if .v2db inventory files are found on pre-V1R12 release. Earlier releases of Infoprint Server are not aware of these V1R12 files and will ignore them and will not merge changes to the existing .db inventory files into the existing .v2db files.
- No parameters
- Interval ONETIME, Severity LOW

Check details – Infoprint Server

(IBMINFOPRINT, ZOSMIGV1R12_INFOPRINT_INVSIZE)

- Verify that there is enough space to create the new (.v2db) inventory files from the old (.db) files when migrating to V1R12.
- No parameters
- Interval ONETIME, Severity MEDIUM

Check details – BCP Supervisor

(IBMSUP,SUP_LCCA_ABOVE_16M)

- Verifies that the residency for the LCCA control block (macro IHALCCA) is defined to a given RMODE
- In z/OS V1R12 the default residency for the LCCA changes to RMODE 31 (above 16M) to leave more storage available below 16M. Users might have explicitly asked in the past for RMODE 24 (below 16M) via DIAGxx statement CBLOC VIRTUAL24(IHALCCA)
- Parameter 'CBLOC({24|31})') indicates what to expect for the LCCA RMODE. Default is 31 for RMODE 31.
- Interval ONETIME, Severity LOW
- Also, by APAR OA32015 for R10,R11: (IBMSUP,ZOSMIGV1R12_SUP_LCCA_ABOVE_16M)

Check details – Reconfiguration

(IBMRCF,RCF_LCCA_ABOVE_16M)

- Verifies that the residency for the PCCA control block (macro IHAPCCA) is defined to a given RMODE
- In z/OS V1R12 the default residency for the PCCA changes to RMODE 31 (above 16M) to leave more storage available below 16M. Users might have explicitly asked in the past for RMODE 24 (below 16M) via DIAGxx statement CBLOC VIRTUAL24(IHAPCCA)
- Parameter ‘CBLOC({24|31})') indicates what to expect for the PCCA RMODE. Default is 31 for RMODE 31.
- Interval ONETIME, Severity LOW
- Also, by APAR OA32015 for R10,R11: (IBMRCF,ZOSMIGV1R12_RCF_LCCA_ABOVE_16M)

Summary

- **R12 check-writer support**
  - Metal C
  - No Message Table
- **R12 installation support**
  - SuperUser
  - SDSF Health Check History
- **Pre-R12 check-writer support**
  - Persistent Data
  - IOSSPOF service
- **Other RedPaper goodies**
- **R12 new checks**

Publications

- IBM Health Checker for z/OS User's Guide (SA22-7994)
  - Includes an inventory of all health checks known at publication time
  - For an updated list check URL http://www-03.ibm.com/systems/z/os/zos/hchecker/check_table.html
- “Exploiting the Health Checker for z/OS infrastructure”
  - RedPaper 4590 http://www.redbooks.ibm.com/abstracts/redp4590.html?Open
- Metal C Programming Guide and Reference (SA23-2225)
  - http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/EZ2ZBK0H?filter=metal&SUBMIT=Find

*