z/OS Basics: ABEND and Recovery
(All You Need to Know to Write Your First ESTAE)

Vit Gottwald
CA Technologies

August 3, 2010
Session Number 8017
Agenda

- Introduction
  - Basic Hardware Terms
  - Instruction Execution Loop
  - Interrupts
- Recovery
  - Program Error
  - Recovery/Termination Manager
  - ESTAE
  - z/OS Control Blocks
  - Special Considerations
- References
Basic terms

- Storage
  - Programs
  - Data
  - *Low Core* (first 8K of storage)
- CPU
  - 16 General Purpose Registers
  - Program Status Word (instruction pointer)
- Instruction
  - Operation code
  - Operands
  - Length
Instruction Execution Loop

- Sequential

![Diagram showing the execution loop with steps:
  - Fetch Instruction pointed to by PSW into CPU
  - Update PSW to point to The Next Sequential Instruction
  - Execute the instruction]

- How does the CPU know the instruction length?
  - First two bits of operation code
    - 00 – instruction is 2 bytes long
    - 01 or 10 – instruction is 4 bytes long
    - 11 – instruction is 6 bytes long
Instruction Execution Loop

- **Branch**

  - Fetch Instruction pointed to by PSW into CPU
  - Update PSW to point to The Next Sequential Instruction
  - Branch type instruction?
    - YES: Update PSW from the instruction
    - NO: Execute the instruction

- Branch type instructions replace the instruction address in PSW
Instruction Execution Loop

- Branch & Interrupt

1. Fetch Instruction pointed to by PSW into CPU
2. Update PSW to point to The Next Sequential Instruction
3. Branch type instruction?
   - YES: Update PSW from the instruction
   - NO: Execute the instruction
4. Is there an interrupt to service?
   - YES: Hardware INTERRUPT handling
   - NO: Repeat process

SHARE in Boston
What does the hardware do?

- Save into Low Core
  - Current PSW
  - PSW extension
    - Interrupt code
    - Instruction Length Code (ILC)
  - TEA
  - BEAR discussed later

- Load from Low Core
  - New PSW assigned to the type of interrupt that occurred
Interrupts

- Each interrupt type has its own fields in Low Core
  - old-PSW
  - new-PSW
- First Level Interrupt Handler (FLIH)
  - Routine pointed to by instruction address in new-PSW
- Interrupt types
  - Restart, External, Machine Check, I/O
  - SVC
  - Program Check
    - CPU recognized problem in execution of an instruction
    - Categorized by Program Interruption Code (PIC)
## Program Interruption Code (PIC)

<table>
<thead>
<tr>
<th>PIC</th>
<th>Reason</th>
<th>Type of instruction ending</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Operation</td>
<td>suppressed</td>
</tr>
<tr>
<td>0002</td>
<td>Privileged operation</td>
<td>suppressed</td>
</tr>
<tr>
<td>0003</td>
<td>Execute</td>
<td>suppressed</td>
</tr>
<tr>
<td>0004</td>
<td>Protection</td>
<td>suppressed or terminated</td>
</tr>
<tr>
<td>0005</td>
<td>Addressing</td>
<td>suppressed or terminated</td>
</tr>
<tr>
<td>0006</td>
<td>Specification</td>
<td>suppressed or completed</td>
</tr>
<tr>
<td>0007</td>
<td>Data</td>
<td>suppressed, terminated or completed</td>
</tr>
<tr>
<td>0008</td>
<td>Fixed-point overflow</td>
<td>completed</td>
</tr>
<tr>
<td>0009</td>
<td>Fixed-point divide</td>
<td>suppressed or completed</td>
</tr>
<tr>
<td>000A</td>
<td>Decimal overflow</td>
<td>completed</td>
</tr>
<tr>
<td>000B</td>
<td>Decimal divide</td>
<td>suppressed</td>
</tr>
<tr>
<td>000C</td>
<td>HFP exp. overflow</td>
<td>completed</td>
</tr>
<tr>
<td>000D</td>
<td>HFP exp. underflow</td>
<td>completed</td>
</tr>
<tr>
<td>000E</td>
<td>HFP significance</td>
<td>completed</td>
</tr>
<tr>
<td>000F</td>
<td>HFP divide</td>
<td>suppressed</td>
</tr>
<tr>
<td>0010</td>
<td>Segment translation</td>
<td>nullified</td>
</tr>
<tr>
<td>0011</td>
<td>Page translation</td>
<td>nullified</td>
</tr>
</tbody>
</table>

For more PICs see, SA22-7832-07, Chapter 6, Figure 6-1 Interruption Action
Program error

• Hardware detected (Program Check)
  • FLIH receives control and decides whether the program check is an error (e.g. PIC 11 - page fault - is not always a program error)
  • If the P.C. is considered an error, FLIH passes control to RTM
  • Results in 0Cx ABENDs

• Software detected
  • Either a z/OS component or a user program discover a problem and decide to terminate abnormally (call ABEND macro)
  • The call of ABEND macro is an entry to RTM
  • Typically the ABEND code is in the form xNN
    • NN - SVC hex number of the z/OS service detecting the problem
    • e.g. x13 is a group of ABENDs related to open processing
Recovery/Termination manager (RTM)

- Receives control early after the discovery of a *program error* (or when a program ends normally)
- Passes control to appropriate *recovery routine* (if present)
- If recovery not successful and either of
  - //SYSUDUMP, //SYSABEND, or //SYSMDUMP DD present, requests documentation of the error by calling z/OS dump services (SNAP macro)
- Handles the final *termination* of the program
  - Closing any open datasets
  - Freeing memory
  - Releasing ENQs
Recovery routine

- Responsible for
  - Fixing the error and giving the failing program another chance (retry)
  - Documenting the error, cleaning up resources, and continuing with termination process (percolate)

- Two basic types
  - ESPIE – to handle Program Checks with PIC 1–F hex
  - ESTAE-like – to handle ABENDs (Program Checks are special case)
Extended Specify Task Abnormal Exit (ESTAE)

- Established through ESTAE macro
- At entry receives pointers to
  - Parameter specified by the user at ESTAE macro call
  - System Diagnostic Work Area (SDWA)
    - Contains the ABEND information
    - May not be available, check if R0 equals OCR hex
- Communicates with RTM via SDWA
  - Read information directly from SDWA
    - SDWAABCC, SDWACRC, SDWAEC1, SDWAILC1, SDWAINC1, SDWAGRSV, SDWAFLGS, SDWATRAN, SDWABEA, ...
  - Write information directly to SDWA
    - SDWASR00 – SDWASR15, ...
Extended Specify Task Abnormal Exit (ESTAE), cont’d

• Communication with RTM via RTM, cont’d
  • SETRP macro
    • Whether to \textit{retry} (RC=4) or \textit{percolate} (RC=0)
    • Specify the \textit{retry address} (RETADDR=)
    • Restore retry registers from SDWA (RETREGS=YES)
    • … all options described in [3]

• SDWA 64 bit extension
  • provided only when SDWALOC31=YES in ESTAE macro call

• Detailed usage in [1], Chapter named “Providing recovery”
  • Not easy to digest at first time (following sample should help)
Translation Exception Address (TEA)

- Location 168-175 in Low Core
- Filled in when page or segment translation occurs (PIC 10 and 11)
- Bits 0-51 contain address of the page we tried to access
- Bits 52-63 are unpredictable
- Provided in SDWA
  - 32 bit portion in SDWATRAN
  - Full 64 bit in SDWA 64 bit extension (SDWATRNE)
Breaking Event Address Register (BEAR)

- 8 bytes long CPU register
- When a branch type instruction is executed, it’s address is placed in the breaking-event-address register
- When a program interruption occurs, the current contents of the BEAR is placed into Low Core location 110-118
- Provided in 64 bit SDWA extension (SDWABEA)
- Priceless for debugging “wild branches”
Very Simple Example

1. Establish an ESTAE
2. Cause a Program Check by branching to FFFFFFFFE hex
3. Recovery routine gets control and sets retry registers:
   • Clear R0 Translation Exception Address to R3
   • ABEND code into R1 Breaking Event Address to R4
   • Reason code into R2
4. Retry
5. Disable the ESTAE
6. Cause an S0C1 ABEND by DC H’0’
   • ESTAE no longer defined -> proceed with termination
   • Register content displayed in the ‘diagnostic dump’ in file 1
Very Simple Sample, cont’d

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY ASMMSP</td>
<td>ENABLE STRUCTURED PROGRAMMING MACROS</td>
</tr>
<tr>
<td>SYSSTATE ARCHLVL=2</td>
<td>USE Z/ARCHITECTURE INSTRUCTIONS</td>
</tr>
<tr>
<td>ASMMREL ON</td>
<td>USE RELATIVE BRANCHING</td>
</tr>
<tr>
<td>SAUTH CSECT</td>
<td>ABOVE THE LINE TO GET BEAR</td>
</tr>
<tr>
<td>SAUTH AMODE 31</td>
<td></td>
</tr>
<tr>
<td>SAUTH RMODE ANY</td>
<td></td>
</tr>
<tr>
<td>STM 14,12,12(13)</td>
<td></td>
</tr>
<tr>
<td>LARL 8,RECOVERY</td>
<td></td>
</tr>
<tr>
<td>LARL 9,RETRY</td>
<td></td>
</tr>
<tr>
<td>ESTAE (8),CT,PARAM=(9),SDWALOC31=YES</td>
<td></td>
</tr>
<tr>
<td>LHI 15,-2</td>
<td>MAX EVEN 31 BIT ADDRESS -&gt; S0C4-11 X SEE BOTH TEA AND BEAR</td>
</tr>
<tr>
<td>BR 15</td>
<td>BRANCH TO HELL (PSW USELESS)</td>
</tr>
<tr>
<td>RETRY DS 0H</td>
<td></td>
</tr>
<tr>
<td>ESTAE 0</td>
<td></td>
</tr>
<tr>
<td>DC H'0'</td>
<td>INVALID OPERATION CODE -&gt; S0C1-1</td>
</tr>
</tbody>
</table>
Very Simple Sample, cont’d

RECOVERY DS 0H

IF CHI,0,EQ,X'0C' Q.SDWA MISSING

* WTO 'SDWA MISSING' may change registers 0,1,14,15
SR 15,15 PERCOLATE
BR 14 RETURN TO RTM=PERCOLATE BY DEFAULT

ENDIF

STM 14,12,12(13) SAVE REGISTERS
LR 3,1 SAVE POINTER TO SDWA
USING SDWA,3 MAP SYSTEM DIAGNOSTIC SAVE AREA
...

SETRP RC=4,RETADDR=(2),WKAREA=(3),RETREGS=YES
DROP 3
LM 14,12,12(13) LOAD REGISTERS
BR 14 RETURN TO RTM

IHASDWA GENERATE SDWA DSECT
END SAUTH END ASSEMBLY
Very Simple Sample, cont’d

*-----------------------------------------------*
SR  0,0
ST  0,SDWASR00
MVC  SDWASR01,SDWAABCC  SAVE ABEND CODE IN R1
MVC  SDWASR03,SDWATRAN  SAVE TRANSLATION EXCEPTION ADDRESS
L  4,SDWAXPAD  ADDRESS OF SDWA EXTENSION POINTERS
USING  SDWAPTRS,4

   L  5,SDWASRVP  RECORDABLE EXTENSION
USING  SDWARC1,5
     MVC  SDWASR02,SDWACRC  SAVE REASON CODE
DROP  5
L  6,SDWAXEME  64-BIT EXTENSION
USING  SDWARC4,6
     MVC  SDWASR04,SDWABEA+4  SAVE BREAKING EVENT ADDRESS-31
DROP  6
DROP  4
*-----------------------------------------------*
Some more SDWA fields of interest

- SDWAXPAD – SDWA extension pointers (SDWAPTRS dsect)
- SDWASRVP – address of recordable extension (SDWARC1 dsect)
- SDWAXEME – address of 64-bit extension (SDWARC4 dsect)
- SDWAERRB,on,SDWAPERC – a previous ESTAE percolated
- When your ESTAE gets control make sure whether it is the first one or whether some other ESTAE already percolated !!!
Make sure to

- Establish your recovery routine when your routine gets control from system, exit, or other app.
- Remove the recovery routine before returning to the caller

- Learn more
  - Read “Providing recovery”, especially section “Special considerations” in [1]
  - Learn about TCB and RB chains and how they relate to recovery routines
  - Be careful when dealing with Linkage Stack, see IEALSQRY macro
Multiple ESTAEs

- When your program establishes multiple ESTAEs
- And an ABEND occurs
  1. The most recently defined ESTAE routine gets control
  2. When it decides to percolate, previously defined ESTAE gets control
  3. Ditto
  4. ...

- ESTAE is represented by a STAE Control Block (SCB)
- SCBs form a stack (LIFO) with the newest SCB on the top
- When an ESTAE percolates its SCB is removed from the stack and control is passed to the next on the top
z/OS Dispatcher Control Blocks

TCB

+1C (RBLINK)
+0 (TCBRBP)

PRB

+1C (RBLINK)

SVRB

+1C (RBLINK)

PRB
z/OS Dispatcher Control Blocks
Other Recovery Routine Types

- **ESTAI**
  - Subtask recovery
  - Defined on ATTACH(X) macro with ESTAI= parameter

- **Associated Recovery Routine (ARR)**
  - Recovery for abends in PC routines

- **Functional Recovery Routine (FRR)**
  - Recovery in SRB routines
  - Defined through SETFRR macro
References

- [3] - *MVS Data Areas*
References, cont’d


Please do not forget to fill in the evaluation forms.

Session #8017
z/OS control blocks

• Piece of storage that has a meaning to z/OS
  • Not very verbose, useful if you know what you are looking for and are familiar z/OS (MVS) terminology
z/OS control blocks – PSA, CVT

- Prefix Save Area (PSA)
  - Prefix Area contains several fields that have hard wired addresses in the CPU for interrupt handling. The rest is used by FLIH and various other components of z/OS
  - In z/OS terminology Prefix Area is called Prefixed Save Area
  - Contains pointers to other control blocks
    - Task Control Block (TCB) at offset 21C
    - Address Space Control Block (ASCB) at offset 224
    - Communication Vector Table (CVT) at offset 10

- Communication Vector Table (CVT)
  - Anchor to most if not all z/OS control blocks!
**z/OS control blocks – ASCB, TCB**

- **Address Space Control Block (ASCB)**
  - Represents single instance of virtual storage to z/OS (recall MVS = Multiple Virtual Storage)
  - Usually one ASCB per Job – XTCB
- **Task Control Block (TCB)**
  - Represents unit of work to z/OS (a *task*)
  - Think of a “task” being a “thread” in PC/UNIX terminology
  - It is an anchor to all resources z/OS allocated on behalf of the task, when TCB is removed, all resources for the task are deallocated
z/OS control blocks - PRB, SVRB

- Request Block (PRB, SVRB, IRB)
  - While TCB represents a unit of work to z/OS, RB represents a particular item we want z/OS to do on behalf of our task
  - When we request a particular program to be run, Program Request Block is created
  - When our program wants to use operating system services, it issues a suitable SVC and a Service Request Block is created
  - External interrupt may generate an asynchronous exit routine to be run (e.g. IRB created for STIMER exit routine)
  - The sequence of the Request Blocks is then called an RB chain, it is chained of a TCB in a reverse order than it was created
RB Chain

- TCB at offset 0 contains a fullword pointer to the most recently created RB
- Each RB points to the previously created RB
- Last RB in the chain (the first created) points back to the TCB
TCB chain

- TCB created by ATTACH macro, DETACH removes
- Program running under a TCB can request further TCBs to be created -> multi-threaded application
- Here the mother task a) attached three daughter tasks (subtasks) b), c), and d) in the respective order

TCBLTC (+88) field points to the subtask the current TCB attached last
TCBOTC (+84) –not shown on picture- points to the parent task
TCBLTC (+80) – points to the task attached previously by parent task
How does RTM receive control?

- Through an ABEND macro call (SVC 13 - 0A0D)
  - Terminates either current TCB or the job step TCB in the current address space

- Through a CALLRTM macro call
  - TYPE=ABTERM
    - a “super” version of ABEND
    - Allows to terminate a (TCB=) in current or other address space
  - TYPE=MEMTERM
    - Terminates an address space without giving control to task level recovery routines and resource managers
Recovery/Termination macros

- CALLRTM
  - TYPE=ABTERM is used by CANCEL operator command
  - TYPE=MEMTERM is used by the FORCE operator command
  - You definitely want to stay away from it, supervisor state and key 0 is required to do a CALLRTM
Recovery/Termination macros

- **ABEND**
  - Generates an SVC 13 (0A0D)
  - Also has a branch entry
  - Allows to specify
    - ABEND code (12 bits) - separate values for System/User ABEND
    - Reason code (RETURN=, 32 bits) - passed to recovery routines
    - Dump options
      - *DUMP* – request a dump
      - *DUMPOPT* – parm. list for the SNAP macro
    - Scope of the ABEND
      - *STEP* – if specified, the job step TCB is terminated, if not specified, the default is to terminate the current TCB
RTM1 and RTM2

- RTM is composed of two parts
  - RTM1 aka “System Level RTM”
  - RTM2 aka “Task Level RTM”
- RTM1
  - Entered via CALLRTM (e.g. from FLIH for an erroneous P.C.)
  - Runs under the environment of the failing program
  - ESPIE registers with RTM1 – low overhead recovery routine
- RTM2
  - Entered via ABEND macro call either from RTM1 or directly
  - Runs as an z/OS subroutine (RB created – 0A0D)
  - ESTAE registers with RTM2 (another RB created when called)
ESTAE macro

• Assume you are writing your first ESTAE routine for your very simple program to recover from a B37 system ABEND

• You will use
  ESTAE EXIT_ADDR, CT, PARAM=PARM_LIST
  • EXIT_ADDR – address of the recovery routine
  • PARM_LIST - parameter list passed to the recovery routine when it is invoked by RTM
  • CT – create as opposed to OV - override an existing ESTAE
Virtual Storage

- Virtual storage
  - Introduced in S/370 in early 1970’s
  - Each “application” (address space) can use the full range of addresses available on the architecture independently of all other applications
  - Implemented in hardware via Dynamic Address Translation

- VIRTUAL ADDRESSES translated into REAL ADDRESSES
z/Architecture Virtual Storage

Virtual address space 1

Real address space

Virtual address space 2
z/Architecture Virtual Storage

Virtual address space 1

Real address space

Virtual address space 2
z/Architecture Virtual, Real, Absolute

- How to handle this with multiple CPUs?

- Prefix register
  - 64 bits, bits 0-32 are always 0
  - Used for assigning a range of real addresses 0-1FFF to a different block in *absolute* storage for each CPU
  - The mechanism is called *Prefixing*, the storage *Prefix Area*
z/Architecture Prefixing

Real Address
CPU 1

Absolute
Address

Real Address
CPU 2
z/Architecture Prefixing

Say Prefix register value in CPU1 is 6000, then

- Real Addresses 1-1FFF are translated to Absolute Addresses 6000-7FFF
- Real Addresses 6000-7FFF are translated to Absolute Addresses 1-1FFF
z/Architecture Prefixing

Real Address
CPU 1

Prefix register value, CPU 1

Absolute Address

Prefix register value, CPU 1

Real Address
CPU 2

Prefix register value, CPU 2
General Purpose Registers

- 16 General (Purpose) Registers (GPR 0 – 15)
  - 64 bits numbered 0 (MSB) – 63 (LSB)
  - Integer arithmetic
  - Address generation/calculation
### z/Architecture Program Status Word

<table>
<thead>
<tr>
<th>OR000TIE</th>
<th>Key</th>
<th>0MWP</th>
<th>AS</th>
<th>CC</th>
<th>Prog Mask</th>
<th>00000000</th>
<th>EA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>31</td>
</tr>
</tbody>
</table>

```
B A
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Instruction Address

51
ESA/390 Program Status Word

- So far z/OS doesn’t support execution of instructions above the 2GB bar (no room in current control blocks to save all 8 bytes of the instruction address upon an interrupt)
- Usually we still deal with the ESA/390 style PSW in dumps and within various z/OS control blocks

<table>
<thead>
<tr>
<th>0</th>
<th>R</th>
<th>0</th>
<th>0</th>
<th>T</th>
<th>I</th>
<th>E</th>
<th>O</th>
<th>X</th>
<th>Key</th>
<th>1</th>
<th>MWP</th>
<th>AS</th>
<th>CC</th>
<th>Prog Mask</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BA</th>
<th>Instruction Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>
Types of Instruction Ending

• Completion
  • Successful completion or partial completion (for interruptible instructions at a unit of work boundary – CC=3)
  • PSW points to the next sequential instruction

• Suppression
  • As if the instruction just executed was a no-operation (NOP)
  • contents of any result fields, including condition code are not changed
  • PSW points to next sequential instruction
Types of Instruction Ending, cont’d

- Nullification
  - Same as Suppression but
  - PSW points to the instruction just executed

- Termination
  1) causes the contents of any fields due to be changed by the instruction to be unpredictable (some may change, other not)
  - The operation may replace all, part, or none of the contents of the designated result fields and may change the condition code
  - PSW points to the next sequential instruction

1) For detailed description see SA22-7832-07, Chapter 5, Type Of Instruction Ending
Termination

- Releasing all resources acquired by the task being terminated
- RTM calls *Resource Managers* to do the actual cleanup
  - Closing any open datasets
  - Freeing memory
  - Releasing ENQs
  - ...
- Performed for both normal and abnormal program end