IBM Software Group

12335: Enterprise PL/I 4.3 Highlights

February 2013

Rational. software

Go to IBM

Peter Elderon

elderon@us.ibm.com

Enterprise 4.3

- performance
- middleware support
- usability

performance

Rational software

Go to IBM

zEnterprise EC12 exploitation

Rational software

Go to IBM

EC12

- **The new zEnterprise EC12 hardware was introduced in August**
- **Enterprise PL/I 4.3 provides immediate and significant exploitation of the new hardware under the ARCH(10) option**
- **Specifying ARCH(10) will cause your code to be tuned for the EC12**

Decimal-Floating-Point Zoned-Conversion Facility

- **This facility adds a new set of instructions for converting between decimal-floating-point (DFP) and zoned decimal**
- **Few customers are currently using DFP**
- **So the usefulness of these new instructions might seem limited**
- **But the compiler can exploit these for you – even in programs that use no floating-point data!**
- **But first a little review:**

Terminology review: zoned decimal

- **Zoned decimal data consists of bytes where the leftmost 4 bits are called the zone bits and the rightmost 4 bits are the decimal or numeric bits.**
- **Most commonly, these are the byte values representing the numbers 0-9**
- **Zoned decimal data is suitable for input, editing, and output of numeric data in human-readable form**
- **There are no arithmetic instructions that operate directly on zoned decimal**
- **Zoned decimal is represented in PL/I by the PICTURE data type**

Terminology review: floating-point

- **A finite floating-point number has three components: a sign bit, an exponent, and a significand.**
- **Its unsigned value is the product of the significand and a radix (or base) value raised to the power of the exponent**
- **It can be used to represent data such as Avogadro's number (6.022E23)**
- **Floating-point numbers are very useful in scientific calculations**

Terminology review: floating-point

- **For many years, IBM mainframes supported floating-point data only with a radix of 16, and such data is called hexadecimal-floating-point**
- **The first IEEE standard for floating-point data used a radix of 2, and such data is called binary-floating-point. IBM mainframes have supported this representation for over 10 years**
- **Both binary- and hexadecimal-floating-point can lead to problems when used to represent decimal data (for example, both represent the value one-tenth as an approximation)**
- **Decimal-floating-point (DFP) uses a radix of 10 is part of the latest IEEE standard and avoids these problems**

Terminology review: floating-point

- **Floating-point data is represented in PL/I by the FLOAT data type**

- **Depending on the BINARY/DECIMAL attribute and the settings of the DFT and FLOAT compiler options, data with any of the 3 radices may be represented in PL/I:**

	DFT(HEX) FLOAT(NODFP)	DFT(HEX) FLOAT(DFP)	DFT(IEEE) FLOAT(NODFP)	DFT(IEEE) FLOAT(DFP)
FLOAT BIN	radix = 16	radix = 16	radix = 2	radix = 2
FLOAT DEC	radix = 16	radix = 10	radix = 2	radix = 10

Decimal-Floating-Point Zoned-Conversion Facility

- **There are no instructions that perform arithmetic on zoned decimal or that support converting zoned decimal to binary integer**
- **But for many years there have been instructions to convert from zoned decimal to packed decimal (for which there are nice arithmetic and conversion instructions)**
- **And there are instructions to convert back from packed to zoned**
- **But: some of these instructions are slow**
- **Also: packed data cannot be held in registers, and that hinders optimization**

Decimal-Floating-Point Zoned-Conversion Facility

- **This new facility in the zEnterprise EC12 hardware adds instructions to convert from zoned decimal to DFP (and back)**
- **And there are already arithmetic instructions that operate on DFP as well as instructions to convert between DFP and binary integer**
- **Also: DFP data can be held in registers, and that helps optimization**
- **These new instructions will clearly help in programs that use PICTURE and DFP data**

Example: Picture to Decimal-Floating-Point

- **So, for example, when given this code to convert PICTURE to DFP**

```
*process float(dfp);

 pic2dfp: proc( ein, aus ) options(nodescriptor);

   dcl ein(0:100_000) pic'(9)9' connected;
   dcl aus(0:hbound(ein)) float dec(16) connected;
   dcl jx  fixed bin(31);

   do jx = lbound(ein) to hbound(ein);
     aus(jx) = ein(jx);
   end;
 end;
```

Example: Picture to Decimal-Floating-Point

- **Under ARCH(9), the heart of the loop consists of these 17 instructions**

```
0060  F248  D0F0  F000      PACK   #pd580_1(5,r13,240),_shadow4(9,r15,0)
0066  C050  0000  0035      LARL   r5,F'53'
006C  D204  D0F8  D0F0      MVC    #pd581_1(5,r13,248),#pd580_1(r13,240)
0072  41F0  F009            LA     r15,#AMNESIA(,r15,9)
0076  D100  D0FC  500C      MVN    #pd581_1(1,r13,252),+CONSTANT_AREA(r5,12)
007C  D204  D0E0  D0F8      MVC    _temp2(5,r13,224),#pd581_1(r13,248)
0082  F874  D100  2000      ZAP    #pd586_1(8,r13,256),_shadow3(5,r2,0)
0088  D207  D0E8  D100      MVC    _temp1(8,r13,232),#pd586_1(r13,256)
008E  5800  4000            L      r0,_shadow2(,r4,0)
0092  5850  4004            L      r5,_shadow2(,r4,4)
0096  EB00  0020  000D      SLLG   r0,r0,32
009C  1605                  OR     r0,r5
009E  B3F3  0000            CDSTR  f0,r0
00A2  EB00  0020  000C      SRLG   r0,r0,32
00A8  B914  0011            LGFR   r1,r1
00AC  B3F6  0001            IEDTR  f0,f0,r1
00B0  6000  E000            STD    f0,_shadow1(,r14,0)
```

Example: Picture to Decimal-Floating-Point

- **While under ARCH(10), it consists of just these 8 instructions – and the loop runs more than 4 times faster**

```
0060  EB2F  0003  00DF        SLLK    r2,r15,3
0066  B9FA  202F              ALRK    r2,r15,r2
006A  A7FA  0001              AHI     r15,H'1'
006E  B9FA  2023              ALRK    r2,r3,r2
0072  ED08  2000  00AA        CDZT    f0,#AddressShadow(9,r2,0),b'0000'
0078  B914  0000              LGFR    r0,r0
007C  B3F6  0000              IEDTR   f0,f0,r0
0080  6001  E000              STD     f0,_shadow1(r1,r14,0)
```

Decimal-Floating-Point Zoned-Conversion Facility

- **But, more importantly, the Enterprise PL/I 4.3 compiler exploits this new facility in the zEnterprise EC12 hardware to help programs that don't even use DFP !**

- **For programs that convert PICTURE to FIXED BIN (or the reverse) the compiler has traditionally used packed decimal as an intermediary.**

- **Now it can use DFP instead, and in many cases this is faster**

Example: Picture to Fixed Bin(31)

- **So, for example, when given this code to convert PICTURE to FIXED BIN**

```
pic2int: proc( ein, aus ) options(nodescriptor);

  dcl ein(0:100_000) pic'(9)9' connected;
  dcl aus(0:hbound(ein)) fixed bin(31) connected;
  dcl jx  fixed bin(31);

  do jx = lbound(ein) to hbound(ein);
    aus(jx) = ein(jx);
  end;
end;
```

Example: Picture to Fixed Bin(31)

- **Under ARCH(9), the heart of the loop consists of these 8 instructions**

```
0058  F248  D098  1000    PACK   #pd580_1(5,r13,152),_shadow2(9,r1,0)
005E  C020  0000  0021    LARL   r2,F'33'
0064  D204  D0A0  D098    MVC    #pd581_1(5,r13,160),#pd580_1(r13,152)
006A  4110  1009          LA     r1,#AMNESIA(,r1,9)
006E  D100  D0A4  200C    MVN    #pd581_1(1,r13,164),+CONSTANT_AREA(r2,12)
0074  F874  D0A8  D0A0    ZAP    #pd582_1(8,r13,168),#pd581_1(5,r13,160)
007A  4F20  D0A8          CVB    r2,#pd582_1(,r13,168)
007E  502E  F000          ST     r2,_shadow1(r14,r15,0)
```

Example: Picture to Fixed Bin(31)

- **While under ARCH(10), it consists of 9 instructions and uses DFP in several of them – but since only the ST and the new CDZT refer to storage, the loop runs more than 66% faster**

```
0060  EB2F  0003  00DF     SLLK    r2,r15,3
0066  B9FA  202F           ALRK    r2,r15,r2
006A  A7FA  0001           AHI     r15,H'1'
006E  B9FA  2023           ALRK    r2,r3,r2
0072  ED08  2000  00AA     CDZT    f0,#AddressShadow(9,r2,0),b'0000'
0078  B914  0000           LGFR    r0,r0
007C  B3F6  0000           IEDTR   f0,f0,r0
0080  B941  9020           CFDTR   r2,b'1001',f0
0084  5021  E000           ST      r2,_shadow1(r1,r14,0)
```

Decimal-Floating-Point Zoned-Conversion Facility

- **In converting PICTURE to FIXED BIN, the compiler uses the new CDZT instruction that converts zoned-decimal to DFP**
- **In converting from FIXED BIN(31) to PICTURE, the compiler could use the new instruction CZDT that does the reverse**
- **However, our tests showed that this set of instructions performed slightly worse than the old set**
- **This is another example of the strength of the compiler: it will exploit new instructions exactly when they help you - and as another example of this, consider conversions of UNSIGNED FIXED BIN(32) to PICTURE**

Example: Unsigned Fixed Bin(32) to Picture

- **So, when given this code to convert UNSIGNED FIXED BIN to PICTURE**

```
uint2pic: proc( ein, aus ) options(nodescriptor);

  dcl ein(0:100_000) unsigned fixed bin(32) connected;
  dcl aus(0:hbound(ein)) pic'(10)9' connected;
  dcl jx  fixed bin(31);

  do jx = lbound(ein) to hbound(ein);
    aus(jx) = ein(jx);
  end;
end;
```

Example: Unsigned Fixed Bin(32) to Picture

- **Under ARCH(9), the heart of the loop consists of these 10 instructions**

```
005C   586E   F000          L       r6,_shadow2(r14,r15,0)
0060   4140   1000          LA      r4,#AMNESIA(,r1,0)
0064   C050   0000   0026   LARL    r5,F'38'
006A   41E0   E004          LA      r14,#AMNESIA(,r14,4)
006E   C067   8000   0000   XILF    r6,F'-2147483648'
0074   4E60   D0A0          CVD     r6,#pd579_1(,r13,160)
0078   D207   D0A8   D0A0   MVC     #pd581_1(8,r13,168),#pd579_1(r13,160)
007E   FA75   D0A8   5000   AP      #pd581_1(8,r13,168),+CONSTANT_AREA(6,r5,0)
0084   D207   D098   D0A8   MVC     _temp1(8,r13,152),#pd581_1(r13,168)
008A   F397   4000   2000   UNPK    _shadow1(10,r4,0),_temp1(8,r2,0)
```

Example: Unsigned Fixed Bin(32) to Picture

- **While under ARCH(10), it consists of only 8 instructions (but uses DFP in several of them), and combined with the facts that only the L and the new CZDT refer to storage and that packed decimal is avoided, the loop runs more than 2 times faster**

```
005C   5851   E000                 L        r5,_shadow1(r1,r14,0)
0060   EB30   0003   00DF          SLLK     r3,r0,3
0066   EB40   0001   00DF          SLLK     r4,r0,1
006C   1E34                        ALR      r3,r4
006E   4110   1004                 LA       r1,#AMNESIA(,r1,4)
0072   B953   0005                 CDLFTR   f0,r5
0076   B9FA   303F                 ALRK     r3,r15,r3
007A   ED09   3000   00A8          CZDT     f0,#AddressShadow(10,r3,0),b'0000'
```

Example: Fixed Bin(63) to Picture

- **Or, when given this code to convert FIXED BIN(63) to PICTURE**

```
quad2pic: proc( ein, aus ) options(nodescriptor);

  dcl ein(0:100_000) fixed bin(63) connected;
  dcl aus(0:hbound(ein)) pic'(18)9' connected;
  dcl jx  fixed bin(31);

  do jx = lbound(ein) to hbound(ein);
    aus(jx) = ein(jx);
  end;
end;
```

Example: Fixed Bin(63) to Picture

- **Under ARCH(9), the heart of the loop consists of these 9 instructions**

```
005E   585E   F000          L       r5,_shadow2(r14,r15,0)
0062   586E   F004          L       r6,_shadow2(r14,r15,4)
0066   41E0   E008          LA      r14,#AMNESIA(,r14,8)
006A   EB05   0020   000D   SLLG    r0,r5,32
0070   1606                 OR      r0,r6
0072   E300   D098   002E   CVDG    r0,_temp1(,r13,152)
0078   EA11   1000   D098   UNPKA   _shadow1(18,r1,0),_temp1(r13,152)
007E   D611   1000   4000   OC      _shadow1(18,r1,0),+CONSTANT_AREA(r4,0)
0084   4110   1012          LA      r1,#AMNESIA(,r1,18)
```

Example: Fixed Bin(63) to Picture

- **While under ARCH(10), it consists of 13 instructions (and uses DFP in several of them), but since only the L and the new CZXT refer to storage and since there are no packed decimal references, the loop runs more than 2.5 times faster**

```
005C  5801  E000            L      r0,_shadow1(r1,r14,0)
0060  EB4F  0004  00DF      SLLK   r4,r15,4
0066  EB5F  0001  00DF      SLLK   r5,r15,1
006C  5861  E004            L      r6,_shadow1(r1,r14,4)
0070  4110  1008            LA     r1,#AMNESIA(,r1,8)
0074  1E45                  ALR    r4,r5
0076  B9FA  4042            ALRK   r4,r2,r4
007A  EB00  0020  000D      SLLG   r0,r0,32
0080  1606                  OR     r0,r6
0082  B3F9  0000            CXGTR  f0,r0
0086  EB00  0020  000C      SRLG   r0,r0,32
008C  ED11  4000  00A9      CZXT   f0,#AddressShadow(18,r4,0),b'0000'
0092  A7FA  0001            AHI    r15,H'1'
```

Decimal-Floating-Point Zoned-Conversion Facility

- **To summarize some of the lessons from these examples:**
  - **A longer set of instructions may be faster than a shorter set**
  - **Reducing storage references helps performance**
  - **Eliminating packed decimal instructions can help performance**
  - **Using decimal-floating-point may improve your code's performance even in program's that have no floating-point data**
  - **The 4.3 PL/I compiler knows when these new ARCH(10) instructions will help and will exploit them appropriately for you**

Other performance enhancements

Rational. software

Go to IBM

VERIFY and SEARCH improved

- When the second argument to VERIFY and SEARCH is a single character, then the code for them will now be inlined
- Previously this was done only when they had 2 arguments, but not 3
- This makes it easy to write well-performing code that parses a blank-delimited string
- For example, this code will now perform much better than previously when both the VERIFY and SEARCH functions were done by library calls

VERIFY and SEARCH improved

```
argcount = 0; kx = 1;
findArgs:
do loop;
  argcount += 1;
  jx = verify(x,' ',kx); /* find next non-blank */
  if jx = 0 then do;
    argvals(argcount) = substr(x,kx);
    leave findArgs;
  end;
  kx = search(x,' ',jx); /* find blank after that */
  if kx = 0 then do;
    argvals(argcount) = substr(x,jx);
    leave findArgs;
  end; else
    argvals(argcount) = substr(x,jx,kx-jx);
 end;
```

More conversions from BIT to CHAR inlined

- **The compiler now handles more conversions of BIT to CHAR by generating inline code (rather than a call to a library routine)**
- **In particular, if the BIT source has length 8 or less and a known offset, then the conversion will be inlined**
- **Previously this was done only if it had length 1 and a known offset**
- **Of particular importance here is that now BIT(8) to CHAR will be inlined**

More conversions of BIT to WCHAR inlined

- **The compiler now also handles more conversions of BIT to WIDECHAR by generating inline code (rather than a call to a library routine)**

- **In particular, if the BIT source has length 8 or less and a known offset, then the conversion will be inlined**

- **Previously no conversions of BIT to WIDECHAR were inlined**

Faster code for TRIM of FIXED DEC

- **The TRIM function is very useful in preparing numbers to be inserted into strings and messages**

- **To make it more useful, the code generated for TRIM of FIXED DECIMAL has been improved so that it performs better**

Middleware improvements

SQL ONEPASS

- **The SQL preprocessor now supports the ONEPASS option**
- **As was true in previous releases, this option requires that host variables be declared before they are used**
- **It has no effect on the number of times the preprocessor reads the source – it always reads it only once**
- **And no effect on the performance of the preprocessor**

SQL statement display

- **When EXEC SQL statements are shown in the listing, they will now reflect their original source formatting (rather than just the tokenized form of the statement)**
- **This preserves comments and makes them easier to read**
- **This change was also made to the 4.2 preprocessor via PTF**

SQL and restricted expressions

- **The SQL preprocessor will now honor the use of some restricted expressions in host variable declarations to define the bounds of an array or the length of a string**

- **But the restricted expression must contain only**
  - **Integers (either literals or previously declared FIXED BIN VALUEs)**
  - **One of the built-in functions INDICATORS, HBOUND, LENGTH, and MAXLENGTH**
  - **An add or subtract operator applied to such an expression**
  - **A multiply operator applied to such an expression**
  - **A prefix operator applied to such an expression**

- **So, in this example, B could now be used as a host variable**
  - **Dcl A char(20), B char(2*(length(A)+3);**

SQL and LIKE

- **The SQL preprocessor will now recognize host variables that are part of structures declared with the LIKE attribute**

- **The preprocessor will handle LIKE in exactly the same way as the compiler and with exactly the same restrictions**

SQL and DEPRECATE

- **The SQL preprocessor now has its own DEPRECATE option**
- **It currently supports only a STMT suboption with only these suboptions**
  - **EXPLAIN**
  - **GRANT**
  - **REVOKE**
  - **SET_CURRENT_SQLID**
- **So specifying PPSQL(DEPRECATE(STMT(GRANT, REVOKE))) would cause any compilation of a SQL program using EXEC SQL GRANT or EXEC SQL REVOKE to end with some E-level messages**

Increased Usability

Rational. software

Go to IBM

Enhanced UTF-8 support

Rational. software

Go to IBM

New UTF-8 functions

- **This release introduces 3 new functions in support of UTF-8 :**
  - **UTF8**
  - **UTF8TOCHAR**
  - **UTF8TOWCHAR**
- **These allow for easy conversion between CHAR and UTF-8**
- **They also provide the means to create UTF-8 literals and to initialize static variables with UTF-8 data**

UTF8

- **The new UTF8 function converts its argument to its equivalent in UTF-8**
- **In UTF8(x), x can be FIXED, FLOAT, PICTURE, BIT, CHAR, or WCHAR**
- **If x is WCHAR, x is converted to UTF-8 under the assumption that x holds UTF-16 (if not, the generated code will raise the ERROR condition)**
- **Otherwise, the CODEPAGE option specifies the source codepage of x**

UTF8

- **So UTF8('91') is a 2-byte character literal holding '3931'x whether the source is compiled with DFT(EBCDIC) or DFT(ASCII)**

- **And this built-in allows you to create STATIC variables such as**

```
declare months(12) char(10) varying static
                   init(
                        uft8('Januar'),
                        utf8('Februar'),
                        utf8('März'),
                                …
                        utf8('Dezember') );
```

UTF8TOCHAR

- **The new UTF8TOCHAR function converts a CHARACTER expression from UTF-8 to CHARACTER**
- **The CODEPAGE option specifies the target code page**
- **In UTF8TOCHAR(x), x must have CHARACTER type**
- **If x holds invalid UTF-8, the generated code will raise the ERROR condition**

UTF8TOWCHAR

- **The new UTF8TOWCHAR function converts UTF-8 to UTF-16**
- **In UTF8TOWCHAR(x), x must have CHARACTER type**
- **If x holds invalid UTF-8, the generated code will raise the ERROR condition**

Language enhancements

ALLCOMPARE built-in function

- **The new ALLCOMPARE built-in function compares 2 structures on a field-by-field basis**

- **So given**

```
dcl 1 A1, 2 B fixed bin(15) init(0), 2 C fixed bin(15) init(1);
dcl 1 A2, 2 B fixed dec(03) init(0), 2 C fixed dec(03) init(1);
```

- **ALLCOMPARE(A1, A2) would return true (namely ‘1’b)**

- **Note that COMPARE(ADDR(A1), ADDR(A2), STG(A1)) would return false since COMPARE does a binary byte compare of storage**

ALLCOMPARE built-in function

- **The ALLCOMPARE built-in function has an optional third argument that must be a char(2) constant with value ‘EQ’, ‘LE’, ‘LT’, ‘GT’, ‘GE’, or ‘NE’**

- **So given**

```
dcl 1 A1, 2 B fixed bin(15) init(0), 2 C fixed bin(15) init(1);
dcl 1 A2, 2 B fixed dec(05) init(1), 2 C fixed dec(05) init(2);
```

- **ALLCOMPARE(A1, A2) would return false**
- **ALLCOMPARE(A1, A2, ‘LT’) would return true**

- **If the third argument is omitted, ‘EQ’ is assumed**

ASSERT statement

- **The new ASSERT statement resembles the assert statement in Java and the assert function in C/C++**

- **It asserts either that a condition is true or false or whether a statement is unreachable and has 3 formats**

- **ASSERT TRUE(<test-expression>) TEXT(<display-expression>)**
- **ASSERT FALSE(<test-expression>) TEXT(<display-expression>)**
- **ASSERT UNREACHABLE TEXT(<display-expression>)**

- **And in each format, the TEXT clause is optional**

ASSERT statement

- The ASSERT statement is very useful in making your code self-checking and in an easily understood way
- For example, this code
- SELECT; WHEN(account_number > 0); END;
- will raise ERROR if the account_number is bad, but the code is not very understandable – unlike this code
- ASSERT TRUE(account_number > 0)
- TEXT (‘account number is not positive!’);

ASSERT statement

- **When an ASSERT statement fails, the generated code passes these arguments to a routine that you must supply**
  - **The packagename() value**
  - **The procname() value**
  - **The sourceline() value**
  - **The TEXT display-expression value**
- **Your routine can then use any or all of these arguments as desired - for example, inside the compiler we use them as inserts into compiler messages**
- **The routine can then raise ERROR, do a GOTO, force an abend, etc**

ASSERT statement

- **The IGNORE compiler option now accepts ASSERT as a suboption**
- **So, you can have ASSERT statements in your source that would be active in the development version of your application**
- **But then, by compiling with IGNORE(ASSERT), the statements would be compiled out of your production code**

LIKE from LIKE

- **The LIKE attribute is now permitted to specify names of structures that contain fields with the LIKE attribute**
- **BUT: only if those structures are declared first and only if the LIKE reference does not depend on the expansion of a LIKE reference**
- **So, the following declares are valid because A is declared before B and B is declared before C**

```
dcl 1 A, 2 A1 fixed bin;
dcl 1 B, 2 B1 like A;
dcl 1 C, 2 C1 like B;
```

LIKE from LIKE

- **However, the following is not valid**

```
Dcl
   1 A,
     2 A1,
       3 A11 ,
       3 A12 ,
     2 A2,
   1 B like A,
   1 C like B.A1;
```

- **Because the expansion of the LIKE for B occurs only after the compiler has tried to resolve all the LIKE references and so B.A1 is unknown**

The SUPPRESS attribute

- **The SUPPRESS attribute may now be specified on PROCEDURE statements with these suboptions**

- **SUPPRESS(LAXNESTED)**
  -
  - **This will suppress the RULES(NOLAXNESTED) message**

- **SUPPRESS(UNREF)**
  -
  - **This will suppress the RULES(NOUNREF) message**

HANDLE operations

- **The following operations are now supported on HANDLEs**

- **Comparing**
- **Adding to or subtracting from – with sensitivity to the underlying type**
- **Computing the difference – with sensitivity to the underlying type**

- **In the first and third operations, the handles must be to the same type**

- **The sensitivity to the underlying type makes the behavior like that in C. So, for example, adding 1 to a handle increases the associated pointer value by the number of bytes in the underlying structure type**

Miscellaneous

- **The maximum length of WCHAR strings is now 32767 (the same as CHAR)**
- **INOUT and OUTONLY now imply BYADDR**

Miscellaneous user requirements

MSGSUMMARY

- **Under the new MSGSUMMARY option, the compiler includes a message summary near the end of the listing**
- **It is sorted by compiler component and within each component by severity and then by message number.**
- **The summary includes the following information:**
  - **One instance of each message that is produced in the compilation**
  - **The number of times that each message is produced**
- **If the XREF suboption is specified (NOXREF is the default), then**
  - **after each message, the summary lists all the line or statement numbers where the message is issued.**

MSGSUMMARY

- **So, when given this intentionally bad code, the listing will now end with the following summary of the messages produced when compiled with the options PP(SQL,MACRO,CICS),FLAG(I),MSGSUMMARY(XREF)**

```
msgsumm: proc;

  exec sql include sqlca;

  exec cics what now;
  exec cics not this;

  %dcl z0 fixed bin;
  %dcl z1 fixed dec;
end;
```

MSGSUMMARY

```
Summary of Messages

Component Message     Total  Default Message Description

SQL       IBM3250I W      1  DSNH053I DSNHPSRV  NO SQL STATEMENTS WERE FOUND
                             Refs: 4.0
SQL       IBM3250I W      1  DSNH527I DSNHOPTS  THE PRECOMPILER OR DB2 COPROCESSOR
 ATTEMPTED
                             TO USE THE DB2-SUPPLIED DSNHDECP MODULE.
                             Refs: 0.0
SQL       IBM3000I I      1  DSNH4760I DSNHPSRV  The DB2 SQL Coprocessor is using the level
                             2 interface under DB2 V9
                             Refs: 0.0
MACRO     IBM3552I E      2  The statement element %1 is invalid. The statement will be
                             ignored.
                             Refs: 9.0 10.0
MACRO     IBM3258I W      2  Missing %1 assumed before %2.
                             Refs: 9.0 10.0
CICS      IBM3750I S      2  DFH7059I S  WHAT COMMAND IS NOT VALID AND IS NOT TRANSLATED.
                             Refs: 6.0 7.0
Compiler  <none>
```

CASERULES

- **The new CASERULES option allows you to enforce your coding standard for PL/I keywords. It currently has one suboption, KEYWORD, with 4 suboptions**

- **MIXED – permits anything (and is the default)**
- **UPPER – requires all keywords to be in uppercase**
- **LOWER – requires all keywords to be in lowercase**
- **START – requires all keywords to start in uppercase and have only lowercase for any remaining letters**

DEPRECATE

- **The DEPRECATE option has been enhanced with a new STMT suboption**
- **With the STMT option, you can list PL/I statements that you want your programmers not to use (such as, for example, DISPLAY and STOP)**
- **The list of statements that may be deprecated is limited and does not include essential PL/I statements such as IF and SELECT**
- **Since DEPRECATE(STOP) is now supported, the (NO)STOP option of the RULES compiler option has been dropped**

DEPRECATENEXT

- **The new DEPRECATENEXT option is essentially the same as the DEPRECATE option**
- **They both have the same set of suboptions**
- **But instead of producing E-level messages, DEPRECATENEXT produces W-level messages**
- **This makes it much easier to stage the deprecation of language features**

RULES

- **The RULES option has been enhanced with these new suboptions**
  - **(NO)CONTROLLED**
  - **(NO)RECURSIVE**
  - **(NO)LAXNESTED**
- **And RULES(NOLAXIF) will now also flag statements of the form x = y = z if x is not BIT(1) – under the assumption that x, y = z was meant**

RTCHECK

- **The RTCHECK option has been enhanced:**
- **you may now specify NULL370 as a suboption**
- **the compiled code will then check that any pointer used to load or store data is not equal to the old NULL() value**

Im Ueberblick

performance

- zEnterprise EC12 exploitation
- Other performance enhancements
  - VERIFY and SEARCH improved
  - More conversions from BIT to (W)CHAR inlined
  - Faster code generated for TRIM of FIXED DEC

middleware support

- Improved SQL support
  - Restricted expressions allowed in host variables
  - ONEPASS option supported
  - LIKE supported
  - EXEC SQL DECLARE allowed at PACKAGE level
  - DEPRECATE option introduced

usability

- Enhanced UTF-8 support
- ASSERT statement introduced
- ALLCOMPARE built-in allows for structure compares
- MSGSUMMARY option enhances the listing
- CASERULES option enforces naming conventions
- DEPRECATENEXT option allows staged deprecation
- Additional RULES suboptions to control code quality

Learn more at:

- IBM Rational software
- IBM Rational Software Delivery Platform
- Process and portfolio management
- Change and release management
- Quality management
- Architecture management
- Rational trial downloads
- developerWorks Rational
- IBM Rational TV
- IBM Rational Business Partners