

# Strategy and best practice for modern RPG

Barbara Morris  
IBM



# Agenda

- Learn the features of ILE RPG that allow you to write "modern" code
- Learn which old features and customs to avoid
- Learn about how to improve the performance of your calls
- Learn how using service programs can modernize your development

A series of thin, light gray lines radiate from a point on the left side of the slide, creating a fan-like pattern of overlapping triangles and quadrilaterals that extends towards the top right.

**Make sure you pass the right data structure as a parameter**

## Use OPTIONS(\*EXACT) for LIKEDS parameters

The wrong data structures are passed to the parameters. The procedure would receive garbage.

```
dcl-pr check_order ind;  
    order_info likeds(order_t);  
    customer_info likeds(customer_t) const;  
end-pr;  
  
ok = check_order (customer : order);      // Backwards  
ok = check_order (order : customer.name); // Parm 2 is wrong
```

The compiler allows these invalid calls because

- CONST is coded for the "customer\_info" parameter, so the "order" DS can be smaller
- RPG thinks data structures are also character strings

# Use OPTIONS(\*EXACT) for LIKEDS parameters

Code OPTIONS(\*EXACT) for LIKEDS parameters to ensure that the passed parameter is also defined with the same LIKEDS

```
dcl-pr check_order ind;  
    order_info likeds(order_t) options(*exact);  
    customer_info likeds(customer_t) const options(*exact);  
end-pr;  
  
ok = check_order (customer : order);  
ok = check_order (order : customer.name);
```

Now, the compile would fail because the parameters don't meet the OPTIONS(\*EXACT) requirements

- Only the "order" parameter in the second call is correct

## OPTIONS(\*EXACT) may also be good for other parms

You can code OPTIONS(\*EXACT) for ordinary parameters too

- For parameters passed by reference, you can't pass a parameter that is too long
- For parameters passed by VALUE or CONST
  - You can't pass a parameter that might cause overflow
    - \*ISO date to \*YMD date might overflow the year
    - PACKED(9:2) to PACKED(9:3) might overflow
  - You can't pass a parameter that might lose data
    - 123.456 to PACKED(10:2) loses the .006
    - 'Hello' to CHAR(4) or VARCHAR(4) loses 'o'



# Use mixed-case names for your procedures

# Use EXTPROC(\*DCLCASE) for your procedures

Compare these two call stacks:

- Using RPG's default of uppercasing names:

5	QUOCMD	QSYS		/03B3	
	PGMSTK	BMORRIS			_QRNP_PEP_PGMSTK
	PGMSTK	BMORRIS	6		HANDLEORDER
	PGMSTK	BMORRIS	10		CHECKCUSTSTATUS
	PGMSTK	BMORRIS	14		GETCUSTID

- Using a mixed case name:

5	QUOCMD	QSYS		/03B3	
	PGMSTK	BMORRIS			_QRNP_PEP_PGMSTK
	PGMSTK	BMORRIS	6		handleOrder
	PGMSTK	BMORRIS	10		checkCustStatus
	PGMSTK	BMORRIS	14		getCustId





# Use EXTPROC(\*DCLCASE) for your procedures

Using mixed-case names makes your joblog easier to follow too:

```
MCH1211 Escape          40    17/09/05   13:37:55.210768
          From module . . . . . :    PGMSTK
          From procedure . . . . . :    getCustId
          Statement . . . . . :    15
```

# Use EXTPROC(\*DCLCASE) for your procedures

An easy way to get mixed-case names is to use EXTPROC(\*DCLCASE) on your prototype or procedure interface:

```
dc1-proc handleOrder;  
  dc1-pi *n extproc(*dc1case) end-pi;
```

If you add or change EXTPROC:

- Recompile everything that uses the prototype
- Fix your CL to have the mixed-case name  
CALLPRC 'handleOrder'
- Fix your binder source to have the mixed-case  
EXPORT("handleOrder")



# Bullet-proof your /COPY files

# Bullet-proof your /copy files

## Problem:

If you have a date, time, character, UCS-2 or graphic definition in a /copy file, there could be a mismatch between the modules using the /copy file.

Copy file:

```
dc1-pr handleOrder;  
    dueDate date const;    // No format!
```

Program calling the procedure:

```
ct1-opt datfmt(*ymd);  
/copy orderPr  
handleOrder (%date());    // Date error!
```

# Bullet-proof your /copy files

What if you usually have DATFMT(\*MDY) in the H spec?

Module defining the procedure:

```
ctl-opt datfmt(*mdy) ;  
/copy orderPr  
dcl-proc handleOrder export;
```

Program calling the procedure:

```
ctl-opt datfmt(*dmy) ;  
/copy orderPr  
handleOrder (%date());
```

21/02/19

Caller: Feb 21, 2019

Proc: ? 2, 2019

**Bad date exception**

11/02/19

Caller: Feb 11, 2019

Proc: Nov 2, 2019

**Silent error!**

**A silent error is much worse than an exception!**

# Bullet-proof your /copy files

## Problem:

A more subtle issue could be a prototype with a character parameter in a copy file.

Copy file:

```
dc1-pr sendMessage;  
message varchar(100) const;
```

Program calling the procedure:

```
ct1-opt ccsid(*char:500);  
/copy msgPr  
sendMsg ('Hello!');
```

If the job CCSID is 37,  
sendMsg receives  
'Hello|'  
Not exclamation mark!

(Only a few characters are  
different between 37 and  
500)

**You might never discover this in testing!**

# Bullet-proof your /copy files

## Difficult solution:

You **could** add the required keywords to every definition in the copy file.

## Easy solution:

Use the /SET directive to set copy-file defaults for DATFMT, TIMFMT, and the character, UCS-2 and graphic CCSIDS.

The defaults set by /SET will stay in effect until the copy file ends.

```
/SET DATFMT(*ISO) CCSID(*CHAR : *JOB RUN)
```

Also see the /RESTORE directive, if you want to use /SET to temporarily set new defaults for just part of a source member.



# Avoid the RPG cycle



## Linear-main modules

The RPG cycle is rarely needed any more. Most programmers don't really think about how setting on LR or coding a RETURN prevents the calculations from looping, but that's how the cycle works.

```
dsply 'hello';  
*inlr = *off;
```

The compiler requires \*INLR to be set, but it doesn't care what it is set to.

If I run this program, this is how the joblog looks:

```
DSPLY hello (many more before this)
```

```
DSPLY hello
```

```
DSPLY hello
```

```
DSPLY hello
```

```
5 > *SYSTEM/ENDRQS
```

I had to do a SYS-REQ to stop the program from looping forever

```
Last request at level 4 ended.
```

## Linear-main modules

Since 6.1, you can designate one subprocedure to be the main procedure of your program.

The calculations for a subprocedure begin at the beginning and end at the end, so the calculations are "linear" rather than "cyclical".

```
ctl-opt main(sayHello);  
dcl-proc sayHello;  
    dsply 'hello';  
end-proc;
```

If I run this program, this is how the joblog looks:

```
DSPLY hello
```



# Working with partial arrays

# Partial arrays

Most arrays have a maximum number of elements, but not all elements are actually being used.

In the past, programmers had some difficulty keeping the array sorted.

After sorting the array, blanks go to the top:

```
'      '
'      '
' Adams '
' Campbell '
' Jackson '
```

# Partial arrays

The trick that programmers used was to initialize the array to \*HIVAL, so the unused elements would sort to the end.

Now, after sorting the array, \*HIVAL elements go to the end:

```
' Adams          '  
' Campbell      '  
' Jackson       '  
' *****'      (x'FF's)  
' *****'
```

But that trick isn't needed any more. There's no need to sort the entire array if only the first few elements are being used.

## Partial arrays

With %SUBARR, you can limit the sort to only the elements you're using:

```
SORTA %SUBARR(arr : 1 : num_arr_elems);
```

Use %SUBARR to set a subset of the elements:

```
%SUBARR(arr : start : num_per_page) = 'x';
```

To search only some elements of the array, specify the extra parameters for %LOOKUP:

```
index = %LOOKUP('Jack' : arr : 1 : num_arr_elems);
```

# Use varying-dimension arrays instead

Starting in 7.4 ...

You can avoid the problem of using partial arrays by defining your array as varying-dimension (see 'What's new in 7.4')

```
dcl-s array1 char(10) dim(*VAR:10);
```

```
dcl-s array2 char(10) dim(*AUTO:10);
```

# Use varying-length strings



## Varying-length strings

If you have a lot of %TRIM in your code

- Consider using varying-length strings instead
- If you can't change your field to be varying length
  - Copy it (with one %TRIM) to a varying-length field

## Varying-length strings, before

```
dcl-pi *n;
    lib char(10) const;
    file char(10) const;
end-pi;
dcl-s cmd char(100);

cmd = 'DSPPFM ' + %trim(lib) + '/' + %trim(file)
      + ' OUTPUT(*PRINT)';
callp(e) QCMDEXC (cmd : %len(%trim(cmd)));
if %error();
    report ('File ' + %trim(file) + 'does not exist '
           + 'in library ' + %trim(lib) + '.');
endif;
```

# Varying-length strings, after

```
dcl-pi *n;
    libParm char(10) const;
    fileParm char(10) const;
end-pi;
dcl-s lib varchar(10);
dcl-s file varchar(10);
dcl-s cmd varchar(100);

lib = %trim(libParm);
file = %trim(fileParm);

cmd = 'DSPPFM ' + lib + '/' + file
      + ' OUTPUT(*PRINT)';
callp(e) QCMDEXC (cmd : %len(cmd));
if %error();
    report ('File ' + file + 'does not exist '
           + 'in library ' + lib + '.');
endif;
```

# Varying-length strings, compare the calculations

## Before

```
cmd = 'DSPPFM ' + %trim(lib) + '/' + %trim(file)
      + ' OUTPUT(*PRINT)';
callp(e) QCMDEXC (cmd : %len(%trim(cmd)));
if %error();
    report ('File ' + %trim(file) + 'does not exist '
           + 'in library ' + %trim(lib) + '.');
endif;
```

## After

```
cmd = 'DSPPFM ' + lib + '/' + file
      + ' OUTPUT(*PRINT)';
callp(e) QCMDEXC (cmd : %len(cmd));
if %error();
    report ('File ' + file + 'does not exist '
           + 'in library ' + lib + '.');
endif;
```

Not just "nicer",  
but usually faster

# Passing a trimmed parameter

# Passing a trimmed parameter

To pass a parameter that should always be trimmed

**The hard way - always remember to code %TRIM**

```
dcl-pr getFileInfo;  
    file varchar(101) const;  
end-pr;  
getFileInfo (%trim(filename));
```

**The easy way - let OPTIONS(\*TRIM) handle trimming**

```
dcl-pr getFileInfo;  
    file varchar(101) const options(*trim);  
end-pr;  
getFileInfo (filename);
```

# Null-terminated string parameters

To pass a parameter that ends with x'00' (a null-terminator)

**The hard way - manually add the x'00'**

```
dcl-pr openIfsfile extproc('open');  
  file char(101) const;  
  ... more parameters  
end-pr;  
openIfsfile (%trim(filename) + x'00');
```

**The easy way - let OPTIONS(\*STRING) handle the x'00'**

```
dcl-pr openIfsfile extproc('open');  
  file pointer value options(*string);  
  ... more parameters  
end-pr;  
openIfsfile (%trim(filename));
```

# Null-terminated string parameters

Use `OPTIONS *STRING` and `*TRIM` together

```
dcl-pr openIfsfile extproc('open');  
    file pointer value options(*string : *trim);  
    ... more parameters  
end-pr;  
openIfsfile (filename);
```

The passed parameter will be trimmed even if you pass a pointer

```
dcl-s pName pointer inz(%addr(name));  
dcl-s name char(100);
```

```
name = 'myfile.txt          ' + x'00'; // ADD THIS  
openIfsFile (pName); // procedure receives "myfile.txt"
```





# Using data structures for I/O

## Consider using data structures for I/O

A great feature of RPG is that it's **NOT** necessary to use data structures for I/O.

When you do I/O without a data structure, RPG copies the data from the I/O buffer into your program fields or it copies the data from your program fields into the I/O buffer. One field may be a standalone field, and others may be subfields in various data structures.

This is a central feature of RPG. It can be wonderful when the same field is used in two different files.

# Consider using data structures for I/O

**But it's not always the best choice.**

Sometimes it's better to control where the data is read into or written from.

```
read custfile;           // where is the data going?
```

```
read custfile custDs; // custDs gets the data
```



# Names

# Naming conventions

## The most important rule is Be Consistent

- If you use abbreviations, have one standard abbreviation
  - If the standard is that "cvt" is used for "convert", don't name the procedure `convertDate` or `convDate`, name it `cvtDate`
- Use `camelCaseNames` or `underscore_names`, but not both

## Maximize readability

- Name procedures with verb + noun: `placeOrder`, `terminateAccount`.
- Name most variables with noun, or adjective + noun: `quantity`, `yearlyTotal`
- Name indicators with conditions: `isValid`, `orderSuccessful`, `exitKeyPressed` ...

# Use qualified names

```
read ordRec;  
dow not %eof(ord92);  
    ok = checkInventory (city : item_id : quantity);  
    ...  
    read ordRec;  
enddo;
```

Is "city" something that was set by the READ operation? Is "ordRec" really a record in file "ord92"?

```
read ord92.ordRec order;  
dow not %eof(ord92);  
    ok = checkInventory (cust.city  
                        : order.item_id : order.quantity);  
    ...  
    read ord92.ordRec order;  
enddo;
```

It's a bit more code, but it's easy to see where everything comes from

## A bonus if you use qualified file names

The file's fields are available as standalone fields.  
Programmers forget they should only use the data structure

```
dcl-f orders;  
dcl-ds orderDs likerec(orderRec);  
  
read orderRec orderDs; // read into the DS  
if quantity = 0; // BUG, should be orderDs.quantity
```

Qualify the file to avoid even having those standalone fields

```
dcl-f orders qualified;  
dcl-ds orderDs likerec(orders.orderRec);  
  
read orders.orderRec orderDs; // read into the DS  
if quantity = 0; // Compile error. Field doesn't exist
```

Getting the error at compile-time is always better!

## Use alias names

You probably have nice readable alternate names for your files. But you may also have less readable short names.

```
dcl-f orders;  
...  
if ordqty > 0;  
    placeOrder (ordId : cstId : ordqty  
                : splcty : cstcty);           // "splcty" ???
```

Using the alternate (alias) names:

```
dcl-f orders alias;  
...  
if order_quantity > 0;  
    placeOrder (order_id : customer_id : order_quantity  
                : supplier_city : customer_city);
```



# Changing your code to use ALIAS names

Currently, it is not easy to switch a program to use alias names

To use Rdi's renaming feature (Source > Refactor > Rename)

- For each field in the file used in your program
  - Use “Rename” to rename the field to the alias name
  - Click “Continue” when it warns it being an externally-described field
- Repeat for any LIKERECD data structures
- Repeat for any externally-described data structures
- Finally, add the ALIAS keyword to the file

# RFE for RDi to make it easier to add ALIAS to a file

## Vote for RFE 125314

- Sign in at <http://www.ibm.com/developerworks/rfe>
- Click on “Search”
- Check “I want to specify the brand, product family, and product”
  - **Brand:** Servers and Systems Software
  - **Product family:** Programming Languages
  - **Product:** Developer for Power Systems
- Put “ALIAS” in the “Keywords” box
- Click on the “Search” button at the bottom
- Open the RFE and click on “Vote” on the right-hand side



# Defining complex data structures

# Defining complex data structures

Until very recently, there was only one way to define a complex data structure:

1. Define a template for the sub-data structures
2. Define the sub-data structures using LIKEDS

```
dcl-ds emp_t qualified template;  
  name varchar(25);  
  salary packed(7 : 2);  
  is_manager ind;  
end-ds;
```

```
dcl-ds dept qualified;  
  num_emps int(10);  
  emps likeds(emp_t) dim(30);  
end-ds;
```

The more levels of nesting, the more difficult to understand.

# Defining complex data structures

Now, it's possible to directly define the sub data structures.

```
decl-ds dept qualified;  
    num_ems int(10);  
    decl-ds emps dim(30);  
        name varchar(25);  
        salary packed(7 : 2);  
        is_manager ind;  
    end-ds;  
end-ds;
```

# Take advantage of all 63 digits for numeric procedures

# Procedures to handle numeric values of any length

Historically, OPM RPG supported a maximum of 30 digits. ILE RPG supported 31 digits.

Since V5R3, ILE RPG supports 63 digits.

But many programmers still define their "generic" numeric procedures with 31 digits. For example, defining a procedure with a packed(31:9) parameter.

## **Better:**

Take advantage of the entire 63 digit range. If 9 decimal places will always be adequate, define the parameter as packed(63:9).

Or sacrifice a few integer places and increase the accuracy by defining the parameter as packed(63:15).

# How to define a 'binary' field in RPG



## How to define a 4-byte binary for an API?

Historically, RPG only supported a somewhat bizarre form of binary.

RPG forces a 4-byte "binary" field (9 digits and 0 decimals) to have a range of only -999,999,999 to 999,999,999

This means that the "binary" field is basically being treated as a decimal value.

But the true range of a 4-byte binary is

-2,147,483,648 to 2,147,483,647

Sometimes the full range is necessary for an API or an INFDS

## How to define a 4-byte binary for an API?

Since V3R2/V3R6, ILE RPG has supported true integers, both signed and unsigned.

- INT(10) (10i 0) and UNS(10) (10u 0) define true binary
- BINDEC(9) (9b 0) defines "binary decimal"

**RPG's "binary" type should almost always be avoided.**

One **possible** exception is to save space if you **want** a decimal number.

- A 9-digit packed value requires 5 bytes
- A 9-digit binary value only requires 4 bytes

It's very rare now to care about space to that extent.

## Binary fields in externally described files and DS

By default, RPG treats binary fields in externally-described files and data structures as BINDEC fields.

- Fields defined with type 'B' in DDS
- Fields defined as SMALLINT, BIGINT etc in SQL

To have RPG treat these fields as true integer, code `EXTBININT(*YES)` in your H spec.

**Recommendation:** Add `EXTBININT(*YES)` to the set of H spec keywords that are added to every module

# RPG's bizarre default CCSID for character fields

# RPG's bizarre default CCSID for character fields

By default, RPG assumes that alphanumeric fields have the job CCSID.

Actually, that's not quite true.

RPG assumes that the fields have the **mixed SBCS/DBCS CCSID related to the job CCSID**.

- SBCS means “Single byte character set”, characters used in languages like English, Spanish, French, Russian etc.
- DBCS means “Double byte character set”, characters used in languages like Japanese or Chinese.

If your job CCSID is 37, RPG assumes that your alphanumeric fields have CCSID 937 (supporting both English and Chinese).

## Why does this matter?

## RPG's bizarre default CCSID for character fields

Normally, this doesn't matter.

But if you have x'0E' in your field, and that field gets assigned to a UCS-2 field, the x'0E' would be interpreted as a "shift-out" character, and all the data following it would be interpreted as Chinese double byte characters. The UCS-2 field would not have the correct value.

**Solution:** Add CCSID(\*CHAR:\*JOB RUN) to your H spec.

**Recommendation:** Add CCSID(\*CHAR:\*JOB RUN) to the set of H spec keywords that are added to every module.

# Assigning data structures to each other

## Assigning data structures

RPG considers a data structure to be also a character string.

You can assign one data structure to another using EVAL.

```
eval ds1 = ds2;
```

This is fine as long as

- The data structures have identical subfields
- The data structures don't have any null-capable subfields



## Assigning data structures

Rather than using EVAL, use EVAL-CORR ("corresponding").

EVAL-CORR assigns subfield by subfield.

- Subfields that have the same name and compatible data types are assigned. Null indicators are also assigned for null-capable subfields.
- Other subfields are ignored.

Use the EVAL-CORR Summary in the listing to see exactly what is happening for an EVAL-CORR operation.

If two data structures are related by LIKEDS, EVAL-CORR will just copy all the data at once, so there is no need to worry about performance.



# Handling cleanup tasks

## Handling cleanup tasks

To ensure that cleanup tasks are done at the end of a procedure, careful programmers have historically defined a "cleanup" procedure. The cleanup procedure is called

- just before a procedure returns
- from a cancel handler enabled by the CEERTX API

This can be awkward and error-prone

- The cleanup procedure may need access to several variables from the procedure needing the cleanup
- A maintenance programmer may add an early return and forget to add the call to the cleanup procedure

# Handling cleanup tasks

## The solution

Put the cleanup tasks in the ON-EXIT section of the procedure.

```
dc1-proc myproc;  
  ...  
  p = %alloc(1000);  
  ...  
  if not %found;  
    return;  
  endif;  
  ...  
  on-exit;  
    dealloc p;  
end-proc;
```

The ON-EXIT section is **always** run, no matter how the procedure ends.



# Call performance

# Call performance

There are two aspects that can slow down your calls

- Large return values
- Large parameters passed by value

# Improve the performance for large return values

## Use the RTNPARM keyword

- With RTNPARM, the procedure doesn't technically return a value
- Instead, the return value is handled as an extra hidden parameter passed by reference
- For procedures that return large varying-length strings, using RTNPARM can dramatically improve performance

### Make sure

- You change both the prototype and the procedure interface
- You recompile all callers

# Improve the performance of large VALUE parameters

- Use CONST instead
- If you want to modify the parameter within the procedure, copy it to a temporary variable

When a parameter is passed by value, the system actually copies its value twice while passing the parameter.

- A huge improvement over the original Seven Times



# Improve the performance of large VALUE parameters

Consider a VARCHAR(1000) parameter, with the value 'Hello'.

- The system doesn't know that only 7 bytes are important (2 bytes for the length-prefix and 5 bytes for 'Hello')
- So the system copies the entire 1002 bytes (twice) while passing the parameter (2004 bytes)
- If you code CONST instead of VALUE, the system only sees a pointer being passed, so it copies the 16-byte pointer twice (32 bytes)
- If you want to modify the CONST parameter within the procedure
  - When you copy it to a local variable, it will only have to copy 7 bytes (total 32 + 7 bytes)



# Privacy for variables, procedures, and files

# Privacy

There are 5 levels of privacy available to an ILE programmer.

- **Local to a procedure:** the file or variable can only be used within the procedure
- **Global in the module:** the procedure, file, or variable can be used by any procedure in the module
- **Exported from the module:** the variable or procedure can be used by any other module in the same program or service program that imports the variable or calls the procedure
- **Exported from the service program:** the variable or procedure can be used by anything that binds to the service program and imports the variable or calls the procedure
- **Public:** anyone or any program can call a program

# Privacy

The more private something is, the easier it is to change how it is defined or used.

## Rules of thumb:

- Within a module, avoid global variables and global files when possible
- Think carefully about which procedures you export from a service program. If you have a utility module within the service program, export the utility procedures from the **module**, but if they are specific to the service program, don't export them from the **service program**.
- Only use programs for things that need to be programs. Otherwise, use procedures in service programs to restrict them from being called from the command line.



# Use service programs

# Modernize development with service programs

The goal is to be able to easily

- reuse your code without having to copy it to make small modifications
- modify your code without being worried about the impact of your changes

Both goals can be achieved by having many small procedures which only do one thing

Other non-general procedures can combine calls to these procedures to do application-specific things

# Why service programs?

Instead of having separate procedures, you **could** use separate programs, but ...

Why are service programs better?

- Better control of privacy
- Fewer objects in your libraries
  - If you have zillions of small routines, it's "nicer" to have a few service programs than a zillion little programs
- If you have several related procedures, they can be in the same module
  - reduce module initialization time at runtime
  - easier maintenance if similar changes have to be made to several procedures

# Thank YOU

[→ Go to IBM](#)

© Copyright IBM Corporation 2018. All rights reserved.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo, the on-demand business logo, Rational, the Rational logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.



# Special notices

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

# Special notices

IBM, the IBM logo, ibm.com AIX, AIX (logo), AIX 6 (logo), AS/400, BladeCenter, Blue Gene, ClusterProven, DB2, ESCON, i5/OS, i5/OS (logo), IBM Business Partner (logo), IntelliStation, LoadLeveler, Lotus, Lotus Notes, Notes, Operating System/400, OS/400, PartnerLink, PartnerWorld, PowerPC, pSeries, Rational, RISC System/6000, RS/6000, THINK, Tivoli, Tivoli (logo), Tivoli Management Environment, WebSphere, xSeries, z/OS, zSeries, AIX 5L, Chiphopper, Chipkill, Cloudscape, DB2 Universal Database, DS4000, DS6000, DS8000, EnergyScale, Enterprise Workload Manager, General Purpose File System, , GPFS, HACMP, HACMP/6000, HASM, IBM Systems Director Active Energy Manager, iSeries, Micro-Partitioning, POWER, PowerExecutive, PowerVM, PowerVM (logo), PowerHA, Power Architecture, Power Everywhere, Power Family, POWER Hypervisor, Power Systems, Power Systems (logo), Power Systems Software, Power Systems Software (logo), POWER2, POWER3, POWER4, POWER4+, POWER5, POWER5+, POWER6, POWER6+, System i, System p, System p5, System Storage, System z, Tivoli Enterprise, TME 10, Workload Partitions Manager and X-Architecture are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml)

The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

UNIX is a registered trademark of The Open Group in the United States, other countries or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.

Microsoft, Windows and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries or both.

Intel, Itanium, Pentium are registered trademarks and Xeon is a trademark of Intel Corporation or its subsidiaries in the United States, other countries or both.

AMD Opteron is a trademark of Advanced Micro Devices, Inc.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries or both.

TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).

SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECcapc, SPECchpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corp (SPEC).

NetBench is a registered trademark of Ziff Davis Media in the United States, other countries or both.

AltiVec is a trademark of Freescale Semiconductor, Inc.

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc.

InfiniBand, InfiniBand Trade Association and the InfiniBand design marks are trademarks and/or service marks of the InfiniBand Trade Association.

Other company, product and service names may be trademarks or service marks of others.