Enterprise COBOL for z/OS 6.2

Performance Tuning Guide

SC27-9202-00

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 71.

First edition (23 December 2022 update)

This edition applies to IBM® Enterprise COBOL Version 6 Release 2 (program number 5655-EC6) running with the Language Environment® component of z/OS® Version 2 Release 1, and to all subsequent releases and modifications until otherwise indicated in new editions.

You can view or download softcopy publications free of charge in the Enterprise COBOL for z/OS library. Because Enterprise COBOL for z/OS supports the continuous delivery (CD) model and publications are updated to document the features delivered under the CD model, it is a good idea to check for updates once every two months.

It is our intention to update the product documentation for this release periodically, without updating the order number. If you need to uniquely refer to the version of your product documentation, refer to the order number with the date of update.

Contents

Tables

Preface

About this information

This document identifies key performance benefits and tuning considerations when using IBM Enterprise COBOL for z/OS Version 6 Release 2.

First, this document gives an overview of the major performance features and options in Version 6 of the compiler, followed by performance improvements for several specific COBOL statements. Next, it provides tuning considerations for many compiler and runtime options that affect the performance of a COBOL application. Coding techniques to get the best performance are examined next with a special focus on any coding recommendations that have changed when using Version 6.

The final section examines some causes of increased program object size and studies the object size impact of the various new TEST suboptions as well as discussing the related issue of why PDSEs are required for programs compiled using Version 6.

The performance characteristics of Version 5 of the compiler are similar to Version 6. Except where otherwise noted, the information and recommendations in this document are also applicable to Version 5. Because the performance tuning changes required during migration from Version 5 to Version 6 are so small, recommendations and comparisons in this document assume that the reader is migrating from Version 4 of the compiler.

Performance measurements

The performance measurements in this information were made on the IBM z14 system. The programs used were batch-type (non-interactive) applications. Unless otherwise indicated, all performance comparisons made in this information are referencing CPU time performance and not elapsed time performance.

Note: Except where otherwise noted, performance comparisons in this document are measured using microbenchmarks. Each microbenchmark is designed to highlight the compiler's performance improvement in a specific area. Real programs typically use a mix of features. A program will only see a performance improvement from improvements in the compiler if the program spends a significant percentage of its time executing code that is affected by those improvements.

Summary of changes

This section lists the major changes that have been made to this document since Enterprise COBOL for z/OS V6.2. The changes that are described in this information have an associated cross-reference for your convenience. The latest technical changes are marked within >| and |< in the HTML version, or marked by vertical bars (|) in the left margin in the PDF version.

For a complete list of new and improved features in Enterprise COBOL for z/OS 6.2 and COBOL 6.2 with PTFs installed, see What is new in Enterprise COBOL for z/OS 6.2 and COBOL 6.2 with PTFs installed in the *Enterprise COBOL for z/OS What's New*.

Version 6 Release 2 with PTFs installed

- PH05855: `INITIAL`: The new `INITIAL` compiler option allows you to get a program that has initial values in data items each time the program is called, without having to add the `IS INITIAL` clause to the `PROGRAM-ID` paragraph, and without having to use dynamic `CALL` and `CANCEL` statements. (“Using IS INITIAL on the PROGRAM-ID statement or INITIAL compiler option” on page 43)
- PH35100: `INVDATA`: The new `INVDATA` compiler option replaces the deprecated `ZONEDATA` compiler option and provides users fine-grained control over how the compiler generates code to handle `USAGE DISPLAY` and `USAGE PACKED-DECIMAL` data items that contain invalid data. (“INVDATA” on page 21)

- PH41362: NUMCHECK(ZON): NUMCHECK(ZON): LAXREDEF | STRICTREDEF is deprecated but is tolerated for compatibility, and it is replaced by the LAX | STRICT option. ("NUMCHECK " on page 23)

Version 6 Release 2

Architecture exploitation

A new higher level of ARCH(12) is accepted. The new hardware feature, vector packed decimal instructions, is introduced. For more details, see "Architecture exploitation" on page 1.

Enhanced functionality

Changes in IBM Enterprise COBOL for z/OS, Version 6 Release 2 in the *Enterprise COBOL for z/OS Migration Guide* contains a complete list of new and changed functions in Enterprise COBOL V6.2. Some highlights are:

- Support for parsing JSON using the new JSON PARSE statement
- Support for conditional compilation using the new IF, EVALUATE, and DEFINE directives
- Support for controlling the inlining of PERFORM statements using the new INLINE directive (only available in V6.2) and option (also available in V6.1 with service PTFs)
- Support for detecting invalid numeric data using the new NUMCHECK option (also available in V6.1 with service PTFs)
- Support for detecting corruption beyond the end of working storage caused by mismatched parameter blocks using the new PARMCHECK option (also available in V6.1 with service PTFs)

V6.2 continues to support all of the new features introduced in V5 and V6.1.

How to use examples

This information shows numerous examples of sample COBOL statements, program fragments, and small programs to illustrate the coding techniques being described. The examples of program code are written in lowercase, uppercase, or mixed case to demonstrate that you can write your programs in any of these ways.

To more clearly separate some examples from the explanatory text, they are presented in a `monospace font`.

COBOL keywords and compiler options that appear in text are generally shown in SMALL UPPERCASE. Other terms such as program variable names are sometimes shown in *an italic font* for clarity.

If you copy and paste examples from the PDF format documentation, make sure that the spaces in the examples (if any) are in place; you might need to manually add some missing spaces to ensure that COBOL source text aligns to the required columns per the COBOL reference format in the *Enterprise COBOL for z/OS Language Reference*. Alternatively, you can copy and paste examples from the HTML format documentation and the spaces should be already in place.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this information or any other Enterprise COBOL documentation, send your comments to: compinfo@cn.ibm.com.

Be sure to include the name of the document, the publication number, the version of Enterprise COBOL, and, if applicable, the specific location (for example, the page number or section heading) of the text that you are commenting on.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way that IBM believes appropriate without incurring any obligation to you.

Performance measurements

The performance measurements in this information were made on the IBM z14 system. The programs used were batch-type (non-interactive) applications. Unless otherwise indicated, all performance comparisons made in this information are referencing CPU time performance and not elapsed time performance.

Note: Except where otherwise noted, performance comparisons in this document are measured using microbenchmarks. Each microbenchmark is designed to highlight the compiler's performance improvement in a specific area. Real programs typically use a mix of features. A program will only see a performance improvement from improvements in the compiler if the program spends a significant percentage of its time executing code that is affected by those improvements.

Summary of changes

This section lists the major changes that have been made to this document since Enterprise COBOL for z/OS V6.2. The changes that are described in this information have an associated cross-reference for your convenience. The latest technical changes are marked within >| and |< in the HTML version, or marked by vertical bars (|) in the left margin in the PDF version.

For a complete list of new and improved features in Enterprise COBOL for z/OS 6.2 and COBOL 6.2 with PTFs installed, see What is new in Enterprise COBOL for z/OS 6.2 and COBOL 6.2 with PTFs installed in the *Enterprise COBOL for z/OS What's New*.

Version 6 Release 2 with PTFs installed

- PH05855: `INITIAL`: The new `INITIAL` compiler option allows you to get a program that has initial values in data items each time the program is called, without having to add the `IS INITIAL` clause to the `PROGRAM-ID` paragraph, and without having to use dynamic `CALL` and `CANCEL` statements. (“Using IS INITIAL on the PROGRAM-ID statement or INITIAL compiler option” on page 43)
- PH35100: `INVDATA`: The new `INVDATA` compiler option replaces the deprecated `ZONEDATA` compiler option and provides users fine-grained control over how the compiler generates code to handle `USAGE DISPLAY` and `USAGE PACKED-DECIMAL` data items that contain invalid data. (“INVDATA” on page 21)
- PH41362: `NUMCHECK(ZON)`: `NUMCHECK(ZON)`: `LAXREDEF|STRICTREDEF` is deprecated but is tolerated for compatibility, and it is replaced by the `LAX|STRICT` option. (“NUMCHECK ” on page 23)

Version 6 Release 2

Architecture exploitation

A new higher level of ARCH(12) is accepted. The new hardware feature, vector packed decimal instructions, is introduced. For more details, see “Architecture exploitation” on page 1.

Enhanced functionality

Changes in IBM Enterprise COBOL for z/OS, Version 6 Release 2 in the *Enterprise COBOL for z/OS Migration Guide* contains a complete list of new and changed functions in Enterprise COBOL V6.2. Some highlights are:

- Support for parsing JSON using the new JSON PARSE statement
- Support for conditional compilation using the new IF, EVALUATE, and DEFINE directives
- Support for controlling the inlining of PERFORM statements using the new INLINE directive (only available in V6.2) and option (also available in V6.1 with service PTFs)
- Support for detecting invalid numeric data using the new NUMCHECK option (also available in V6.1 with service PTFs)

- Support for detecting corruption beyond the end of working storage caused by mismatched parameter blocks using the new PARMCHECK option (also available in V6.1 with service PTFs)

V6.2 continues to support all of the new features introduced in V5 and V6.1.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this information or any other Enterprise COBOL documentation, send your comments to: compinfo@cn.ibm.com.

Be sure to include the name of the document, the publication number, the version of Enterprise COBOL, and, if applicable, the specific location (for example, the page number or section heading) of the text that you are commenting on.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way that IBM believes appropriate without incurring any obligation to you.

Chapter 1. Why recompile with V6?

Enterprise COBOL V6 includes a number of improvements over Enterprise COBOL V5 and V4. Recompiling your applications will leverage the advanced optimizations and z/Architecture® exploitation capabilities in Enterprise COBOL V6, delivering performance improvements for COBOL on IBM Z. Compared to COBOL V5, the capacity of the COBOL V6 compiler internals are expanded to allow for the compilation and optimization of large programs. You can now use COBOL V6 to compile much larger programs, including COBOL programs that are created by code generators.

Improved performance is delivered through:

- "Architecture exploitation" on page 1
- "Advanced optimization" on page 4
- "Enhanced functionality" on page 5

Architecture exploitation

COBOL V6 continues to support the ARCH option (short for architecture) introduced in COBOL V5. This option exploits new hardware instructions and enables you to get the most out of your hardware investment.

The default setting for ARCH is 7, and other supported values are 8, 9, 10, 11 and 12.

For more information on the facilities available at each level, and the mapping of these ARCH levels to specific hardware models, see *ARCH* in the *Enterprise COBOL for z/OS Programming Guide*.

Each successive ARCH level allows the compiler to exploit more facilities in your hardware leading to the potential for increased performance. To illustrate the benefits from a COBOL application perspective, each ARCH level will be examined in greater detail below.

ARCH(7)

Hardware Feature: Long displacement instructions

Why This Matters For COBOL Performance: COBOL programs often work with a large amount of WORKING-STORAGE, LOCAL-STORAGE, and LINKAGE SECTION data. The long displacement instructions reduce the need for initializing Base Locator cells and tying up registers for this purpose.

Instead, the compiler uses the much larger reach of the long displacement instructions to access 256 times as much data as the standard displacement instructions used exclusively in earlier releases of the compiler.

Hardware Feature: 64-bit "G" format instructions

Why This Matters For COBOL Performance: Whenever BINARY (and its synonym types of COMP and COMP-4) data exceeds nine decimal digits, then the standard instruction set that operates on 32-bit registers can no longer contain the full range of values. In earlier releases of the compiler, this meant converting to another data type (such as packed decimal) or maintaining pairs of 32-bit registers. Both of these solutions add extra overhead and reduce performance.

Even if your BINARY data is declared with 9 or fewer decimal digits, intermediate arithmetic results can exceed nine decimal digits and might require a conversion from a 32-bit to a 64-bit representation. The conversion is needed because some 10 decimal-digit values and all values greater than 10 decimal digits cannot be fully encoded in the 32-bit two's complement representation that is used for BINARY data.

For example, a multiplication of two PIC 9(5) digit BINARY data items results in an intermediate value of 10 digits, as the source operand digit values of five must be added to arrive at the intermediate precision.

When using addition, the intermediate precision is one greater than the highest of the operand precision values. This means that adding a PIC 9(9) value to a PIC 9(1) value results in an intermediate precision of 10 digits.

In V6, the 64-bit "G" format instruction set is used to hold values with up to 19 decimal digits and therefore a type conversion or using pairs of register is no longer needed and performance is dramatically increased. Above the 64-bit limit type conversions are still required, but the performance of these cases has also been improved in V6. See "BINARY (COMP or COMP-4)" on page 51 for a more in-depth discussion of BINARY data and interaction with TRUNC suboptions.

Hardware Feature: 32-bit immediate form instructions for a range of arithmetic, logical, and compare operations

Why This Matters For COBOL Performance: When your application contains binary data involved in arithmetic or compares, particularly if the data exceeds 5 digits, there is considerable opportunity for the compiler to take advantage of these new ARCH(7) 32-bit immediate form instructions.

Compiling with V4 would only allow the use of at most 16 bits worth of immediate data in a single instruction. Any larger values required storing the data in the literal pool, or using multiple other instructions to construct the immediate value in a register.

Both alternatives are less efficient in time and space.

With ARCH(7), immediate values up to 32 bits can be embedded directly in the instruction text with no need to reference the literal pool or generate other instructions that will increase path length. The result is generally smaller and faster code and less literal pool usage (saving the space and any delay in retrieving the data).

ARCH(8)

Hardware Feature: Decimal Floating Point (DFP)

Why This Matters For COBOL Performance: Decimal Floating Point is a natural fit for the packed decimal (COMP-3) and external decimal (DISPLAY) types that are ubiquitous in most COBOL applications. Using ARCH(8) and some OPTIMIZE setting above 0 enables the compiler to convert larger multiply and divide operations on any type of decimal operands to DFP, in order to avoid an expensive callout to a library routine.

This is possible as the hardware precision limit for DFP is much greater than is allowed in the packed decimal multiply and divide instructions.

The overhead of converting to DFP means that it is not suitable for all decimal arithmetic that would not need a library call. However, the ARCH(10) option described later in this section enables much greater use of DFP to improve performance.

Hardware Feature: Larger Move Immediate Instructions

Why This Matters For COBOL Performance: MOVEs of literal data and VALUE clause statements are common in many COBOL applications. Lower ARCH settings and all earlier compiler releases only contained support for moving a single byte of literal data in a single instruction, for example, by using the MVI - Move Immediate Instruction.

Any larger literal data required storing the constant value in the literal pool and using a memory move instruction to initialize the data item. This was less efficient in time and space than being able to embed larger immediate values directly in the instruction text.

With ARCH(8), several new move immediate instruction variants are available to move up to 16 bytes of sign extended data using one or two of these new instructions.

Also, these instructions are exploited regardless of the data type, so binary, internal/external decimal, alphanumeric, and even floating point literals take advantage of these more efficient instructions.

ARCH(9)

Hardware Feature: Distinct Operands Instructions

Why This Matters For COBOL Performance: Updating a data item or index to a new value while retaining the original value occurs frequently in many contexts in a typical COBOL application. One instance is when processing a table as some base value for the table is updated to access the various elements within the table. Under lower ARCH settings or in all earlier compiler releases, almost all instructions available that took two operands to produce a result would also overwrite the input first operand with the result.

For example: a conceptual operation such as:

C = A + B

Implemented with a pre ARCH(9) instruction variant would conceptually have to perform the operation as:

A = A + B
C = A

This means if the original value of A is required in another context, it must first be saved:

T = A
T = T + B
C = T

With ARCH(9), the distinct-operands facility is exploited to take advantage of the new variants of many arithmetic, shift, and logical instructions that will not destructively overwrite the first operand.

So the operation can be implemented in a more straightforward way:

C = A + B

That removes the need for extra instructions to save the original value as it is naturally preserved with the distinct operand instruction form. This feature reduces path length leading to better performance.

ARCH(10)

Hardware Feature: Improved Decimal Floating Point (DFP) Performance

Why This Matters For COBOL Performance: Using ARCH(8) and an OPTIMIZE setting greater than 0 already enables the compiler to make use of DFP to improve performance of packed and external decimal arithmetic in some particular instances. ARCH(10) goes further by adding efficient instructions to convert between DISPLAY (in particular unsigned and trailing signed overpunch zoned decimal) types and DFP.

These ARCH(10) instructions lower the overhead for using DFP for arithmetic on zoned decimal data items and enable the compiler to make much greater use of DFP to improve performance.

Instead of converting zoned decimal data items to packed decimal format to perform arithmetic, the compiler will convert zoned decimal data directly to DFP format and then back again to zoned decimal format after the computations are complete. This generally results in better performance, as the DFP instructions operate on in-register (compared to in-memory) data that is more efficiently handled by the hardware in many cases.

ARCH(11)

Hardware Feature: Improved conversion between packed decimal and Decimal Floating Point (DFP)

Why This Matters For COBOL Performance: At ARCH(10), the compiler is able to convert more efficiently between DISPLAY types and DFP, enabling the compiler to make significant use of DFP to improve performance of packed and external decimal arithmetic. While instructions to convert between packed decimal and DFP existed at ARCH(10), they were inefficient, and the benefit of performing packed arithmetic in DFP was outweighed by the cost of converting packed decimal values to and from DFP.

With ARCH(11), there are new instructions that convert between packed decimal and DFP more efficiently. They lower the overhead for using DFP arithmetic on packed decimal data items, enabling the compiler to make further use of DFP than at ARCH(10).

Instead of performing arithmetic on packed decimal items, the compiler will convert packed decimal data to DFP format and then back again to packed decimal format after the computations are complete. This generally results in better performance, as the DFP instructions operate on in-register (compared to in-memory) data that is more efficiently handled by the hardware in many cases. Due to the more efficient conversion instructions, the benefit of performing arithmetic in DFP outweighs the added cost of converting between packed decimal and DFP instead of performing packed arithmetic directly.

Hardware Feature: Vector Registers

Why This Matters For COBOL Performance: The new vector facility is able to operate on up to 16 byte-sized elements in parallel. With ARCH(11), COBOL V6 is able to take advantage of the new vector instructions to accelerate some forms of INSPECT statements by working with 16 bytes at a time. This can be much faster than operating on 1 byte at a time.

ARCH(12)

Hardware Feature: Vector packed decimal instructions

Why This Matters For COBOL Performance: In ARCH(11) and below, packed decimal arithmetic can only be performed using in-memory data, or by converting the data to Decimal Floating Point (DFP). In ARCH(12), the new vector packed decimal facility enables the compiler to perform native packed decimal arithmetic on data-in registers. This provides the performance advantages of using registers instead of memory, while eliminating the overhead of converting data back and forth between packed decimal and DFP.

Advanced optimization

In addition to deep architecture exploitation, Enterprise COBOL V6 improves performance of your application by employing a suite of advanced optimizations. In V4, the OPTIMIZE option has three settings; however, the kind and number of optimizations enabled in V6 is quite different.

Specifying OPTIMIZE(1) or OPTIMIZE(2) enables a range of general and COBOL specific optimizations.

For example, specifying OPTIMIZE(1) enables optimizations including:

- Strength reduction of complex and expensive operations, such as:
  - Reducing decimal multiply and divide by powers of ten, to simpler and better performing decimal shift operations
  - Reducing binary multiply and divide by powers of two to less expensive shift operations
  - Reducing exponentiation operations with a constant exponent to series of multiplications
  - Refactoring and redistributing arithmetic
- Eliminate common sub expressions, so computations are not duplicated
- Inline out-of-line PERFORM statements to save the branching overhead and expose other optimization opportunities for the surrounding code
- Coalesce sequential stores of constant values to a single larger store to reduce path length
- Coalesce individual loads/stores from/to sequential storage to a single larger move operation to reduce path length and reduce overall object size
- Simplify code to remove unneeded computations
- Remove unreachable code
- Propagate the VALUE OF clause literal over the entire program for data items that are read but never written
- Move nested programs inline to reduce CALL overhead and expose other optimization opportunities for the surrounding code
- Compute constant expressions, including the full range of arithmetic, data type conversions and branches, at compile-time
- Use a better performing branchless sequence for conditionally setting level-88 variables

- Convert some packed and zoned decimal computations to use better performing Decimal Floating Point types
- Perform comparisons of small DISPLAY and COMP-3 items in registers, instead of in memory
- Generate faster code for moves to numeric-edited items
- Use a better-performing sequence for DIVIDE GIVING REMAINDER

When specifying OPTIMIZE(2), all the optimizations above are enabled plus additional optimizations, including:

- Propagate values and ranges of values over the entire program to expose constants and enable simpler sequences of instructions to be used
- Propagate sign values, including the "unsigned" sign encoding, over the entire program to eliminate redundant sign correction
- Allocate global registers for accessing indexed tables, and control PERFORM 'N' TIMES looping constructs to reduce path length
- Remove redundant sign correction operations globally. For example, if a sign correction for a data item in a loop is dominated by one outside of a loop to the same data item, then these sign-correcting instructions in the loop will be removed

Enhanced functionality

In addition to the performance improvements offered on your existing programs through architecture exploitation and advanced optimizations, COBOL V6 also offers enhanced functionality in several areas.

Changes in IBM Enterprise COBOL for z/OS, Version 6 Release 2 in the *Enterprise COBOL for z/OS Migration Guide* contains a complete list of new and changed functions in Enterprise COBOL V6.2. Some highlights are:

- Support for parsing JSON using the new JSON PARSE statement
- Support for conditional compilation using the new IF, EVALUATE, and DEFINE directives
- Support for controlling the inlining of PERFORM statements using the new INLINE directive (only available in V6.2) and option (also available in V6.1 with service PTFs)
- Support for detecting invalid numeric data using the new NUMCHECK option (also available in V6.1 with service PTFs)
- Support for detecting corruption beyond the end of working storage caused by mismatched parameter blocks using the new PARMCHECK option (also available in V6.1 with service PTFs)

V6.2 continues to support all of the new features introduced in V5 and V6.1.

Some highlights in V6.1 are:

- Support for the new ALLOCATE and FREE statements to obtain and release dynamic storage
- Enhancements to the INITIALIZE statement to support FILLER and VALUE clauses
- Support for generating JSON using the new GENERATE JSON statement
- Support for the new VSAMOPENFS and SUPPRESS options
- Enhancements to the SSRANGE option

Some highlights in V5.2 are:

- Enhancements to the following statements for increased compatibility with ISO 2002 COBOL Standard:
  - New keywords LEADING and TRAILING are added to the REPLACING phrase of the COPY statement and the REPLACE statement to improve partial-word replacement operations
  - EXIT statement enhancements to provide a structured way to exit without using a GO TO statement
  - Table SORT statement arranges table elements in a user-specified sequence
- Restored support for AMODE 24 and XMLPARSE(COMPAT)

- Support for the new options of COPYRIGHT, QUALIFY, RULES, SERVICE, SQLIMS, VLR, ZONEDATA, and NUMCHECK
- Enhancements to the ARCH and MAP options
- Remove the SIZE option, and the compiler manages memory dynamically
- New IBM extensions to COBOL:
  - The >>CALLINTERFACE directive specifies the interface convention for CALL and SET statements
  - The enhanced XML GENERATE statement
  - Support for the VOLATILE clause in a data description entry

Some highlights in V5.1 are:

- XML GENERATE enhancements to provide more flexibility and control over the form of the XML document being generated
- XML parsing enhancing improvements through a new special register, XML-INFORMATION
- Support for UNBOUNDED tables and groups to enable top-down mapping of data structures between XML and COBOL applications
- A new set of Unicode intrinsic functions

Chapter 2. Prioritizing your application for migration to V6

In order to prioritize your migration effort to V6, this section describes a number of specific COBOL statements and data type declarations that typically perform better with V5 and V6 versus earlier releases of the compiler. This is not meant to be an exhaustive list, but instead demonstrate some specific known cases where V5 and V6 performs reliably well.

See *Prioritizing your applications* in the *Enterprise COBOL for z/OS Migration Guide* for related information about migrating to maintain correctness of your application.

All performance measurements are compared to running the same program on the same machine level but compiled with V4. In all cases, the V4 programs were compiled with OPTIMIZE(FULL) and other options left at their default settings, except when ARITH(EXTEND) was required for the data that contained more than 18 digits.

COMPUTE

A significant number of COMPUTE, ADD, SUBTRACT, MULTIPLY, DIVIDE statements show improved performance in V6.

Under "**Data types**" in the following examples, *italics* are used to indicate the variant tested for performance, but all data types listed would demonstrate similar performance results.

Larger decimal multiply/divide

Statement: COMPUTE (* | /), MULTIPLY, *DIVIDE*

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: OPT(1 | 2)

Conditions: When intermediate results exceed the limits for using the hardware packed decimal instructions. This occurs at around 15 digits depending on the particular operation.

V4 behavior: Call to runtime routine

V6 behavior: Inline after converting to DFP

Source Example:

```
1 z14v2 pic s9(14)v9(2)
1 z13v2 pic s9(13)v9(2)

Compute z14v2 = z14v2 / z13v2.
```

Performance: V6 is 44% faster than V4

Zoned decimal (DISPLAY) arithmetic

Statement: COMPUTE (+ | - | * | /), ADD, SUBTRACT, MULTIPLY, *DIVIDE*

Data types: DISPLAY

Options: OPT(1 | 2), ARCH(10)

Conditions: In all cases

V4 behavior: Inline using packed decimal instructions

V6 behavior: Inline after converting to DFP

Source Example:

```
1 z12v2 pic s9(12)v9(2)
1 z11v2 pic s9(11)v9(2)
Compute z12v2 = z12v2 / z11v2
```

Performance: V6 is 59% faster than V4

Divide by powers of ten (10,100,1000,..)

Statement: COMPUTE (/), *DIVIDE*

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: Divisor is a power of 10 (e.g. 10,100,1000,...)

V4 behavior: Use packed decimal divide (DP) instruction

V6 behavior: Model as decimal right shift

Source Example:

```
1 p8v2a pic s9(8)v9(2) comp-3
1 p8v2b pic s9(8)v9(2) comp-3

Compute p8v2b = p8v2a / 100
```

Performance: V6 is 52% faster than V4

Multiply by powers of ten (10,100,1000,..)

Statement: COMPUTE (/), *MULTIPLY*

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: Default

Conditions: Multiply is a power of 10 (e.g. 10,*100*,1000,...)

V4 behavior: Use packed decimal multiply (MP) instruction

V6 behavior: Model as decimal left shift

Source Example:

```
1 z5v2 pic s9(5)v9(2)
1 z7v2 pic s9(7)v9(2)

Compute z7v2 = z5v2 * 100
```

Performance: V6 is 64% faster than V4

Decimal exponentiation

Statement: COMPUTE (**)

Data types: COMP-3, *DISPLAY*, NATIONAL

Options: Default

Conditions: In all cases

V4 behavior: Call to runtime routine

V6 behavior: Call to a more efficient runtime routine

Source Example:

```
1  R     PIC  9v9(8) value 0.05.
1  NF    PIC  9(4) value 300.
1  EXP   PIC  9(23)v9(8).

COMPUTE EXP = (1.0 + R) ** NF.
```

Performance: V6 is 95% faster than V4

Decimal scaling and divide

Statement: COMPUTE (/), *DIVIDE*

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: When the divisor value and the decimal scaling cancel out. In the example below, the divide operation necessitates a decimal left shift by 2, and since the divide by 100 is modelled as the decimal right shift by 2, these operations cancel out.

V4 behavior: Use packed decimal shift (SRP) and divide (DP) instructions

V6 behavior: Divide and decimal scaling are cancelled out so instructions equivalent to a simple MOVE operation are generated

Source Example:

```
1 p9v0 pic s9(9) comp-3
1 p10v2 pic s9(10)v9(2) comp-3.

COMPUTE p10v2 = p9v0 / 100
```

Performance: V6 is 89% faster than V4

TRUNC(STD) binary arithmetic

Statement: COMPUTE (+ | - | * | /), *ADD*, SUBTRACT, MULTIPLY, DIVIDE

Data types: BINARY, *COMP*, COMP-4

Options: TRUNC(STD)

Conditions: In all cases

V4 behavior: Use an expensive divide operation to correct digits back to PIC specification

V6 behavior: Only use divide when actually required (in cases of overflow).

Source Example:

```
1 b5v2a pic s9(5)v9(2) comp.
1 b5v2b pic s9(5)v9(2) comp.

COMPUTE b5v2a = b5v2a + b5v2b
```

Performance: V6 is 74% faster than V4

Large binary arithmetic

Statement: COMPUTE (+ | - | * | /), *ADD*, SUBTRACT, MULTIPLY, DIVIDE

Data types: BINARY, *COMP*, COMP-4

Options: TRUNC(STD)

Conditions: Intermediate results exceed 9 digits

V4 behavior: Arithmetic performed piecewise and converted to packed decimal

V6 behavior: Arithmetic performed in 64-bit registers

Source Example:

```
1 b8v2a pic s9(8)v9(2) comp.
1 b8v2b pic s9(9)v9(2) comp.

Compute b8v2a = b8v2a + b8v2b.
```

Performance: V6 is 95% faster than V4

Negation of decimal values

Statement: COMPUTE (-), SUBTRACT

Data types: *COMP-3*, DISPLAY, NATIONAL

Options: Default

Conditions: In all cases

V4 behavior: Treat as any other subtract from zero

V6 behavior: Recognize as a special case negate operation

Source Example:

```
1 p7v2a pic s9(7)v9(2) comp-3.
1 p7v2b pic s9(7)v9(2) comp-3.

Compute p7v2b = - p7v2a.
```

Performance: V6 is 19% faster than V4

Fusing DIVIDE GIVING REMAINDER

Statement: DIVIDE

Data types: *COMP-3*, DISPLAY

Options: OPT(1 | 2)

Conditions: Both the remainder and the quotient of a division are being used

V4 behavior: Separate divide and remainder computations

V6 behavior: Uses a single DP instruction, and recovers both the remainder and the quotient

Source Example:

```
01 A COMP-3 PIC S9(15).
01 B COMP-3 PIC S9(15).
01 C COMP-3 PIC S9(15).
01 D COMP-3 PIC S9(15).

DIVIDE A BY B GIVING C REMAINDER D
```

Performance: V6 is 10% faster than V4

INSPECT

INSPECT REPLACING ALL on 1 byte operands

Statement: INSPECT REPLACING ALL

Data types: PIC X

Options: Default

Conditions: In all cases

V4 behavior: Uses general translate instruction as in all cases

V6 behavior: Handle short cases with a simple test and move

Source Example:

```
1 ITEM PIC X(1)

INSPECT ITEM REPLACING ALL ‘ ‘ BY ‘.’.
```

Performance: V6 is 85% faster than V4

Consecutive INSPECTs on the same data item

Statement: More than one consecutive INSPECT REPLACING ALL on the same data item

Data types: PIC X

Options: OPT(1 | 2)

Conditions: When the compiler can prove, according to the rules of INSPECT, that the optimization will not alter the result.

V4 behavior: Generate separate operations for each INSPECT operation

V6 behavior: Coalesce the separate INSPECTs into a single INSPECT operation

Source Example:

```
1 ITEM PIC X(15)

INSPECT ITEM REPLACING ALL QUOTE BY SPACE.
INSPECT ITEM REPLACING ALL LOW-VALUE BY SPACE.
```

Performance: V6 is 51% faster than V4

INSPECT TALLYING ALL / INSPECT REPLACING ALL

Statement: INSPECT TALLYING ALL / INSPECT REPLACING ALL

Data types: PIC X

Options: ARCH(11)

Conditions: No BEFORE, AFTER, FIRST, or LEADING clause. For REPLACING, the replaced value must have length > 1

V4 behavior: Use regular instructions or runtime calls

COBOL 6 behavior: At ARCH(11), COBOL 6 is able to generate code using the vector instructions introduced in z13. These instructions are able to process up to 16 bytes at a time

Source Example:

```
01 STR PIC X(255).
01 C PIC 9(5) COMP-5 VALUE 0.

INSPECT STR TALLYING C FOR ALL ' '
```

Performance: V6 ARCH(11) is 99% faster than V4

Source Example:

```
01 STR PIC X(255).

INSPECT STR REPLACING ALL 'AB' BY 'CD'
```

Performance: V6 ARCH(11) is 78% faster than V4

MOVE

VALUE clause and initializing groups

Statement: MOVE and VALUE IS

Data types: All types

Options: OPT(1 | 2)

Conditions: Initializing data items with literals

V4 behavior: Series of separate and sequential move instructions

V6 behavior: Coalesces literals and generates fewer move instructions

Source Example:

```
01 WS-GROUP.
  05 WS-COMP3     COMP-3 PIC S9(13)V9(2).
  05 WS-COMP      COMP   PIC S9(9)V9(2).
  05 WS-COMP5     COMP-5 PIC S9(5)V9(2).
  05 WS-COMP1     COMP-1.
  05 WS-ALPHANUM  PIC X(11).
  05 WS-DISPLAY   PIC 9(13) DISPLAY.
  05 WS-COMP2     COMP-2.

Move +0 to WS-COMP5
           WS-COMP3
           WS-COMP
           WS-DISPLAY
           WS-COMP1
           WS-COMP2
           WS-ALPHANUM.
```

Performance: V6 is 69% faster than V4

Moving into numeric-edited data items

Statement: MOVE

Data types: Receiver is numeric-edited

Options: OPT(1 | 2)

Conditions: None

V4 behavior: Uses the ED or EDMK instruction in all cases

V6 behavior: The ED and EDMK instructions, which handle numeric edits, are extremely slow. V6 converts uses of these specialized instructions into a series of other instructions. This is a new optimization technique in V6.

Source Example:

```
01 PRINCIPAL PIC 9(8)V9999 VALUE 1234.1234.
01 AMT-PRINCIPAL PIC $,$$$,$$9.99.

Move PRINCIPAL to AMT-PRINCIPAL.
```

Performance: V6 is 44% faster

SEARCH

SEARCH ALL

Statement: SEARCH ALL

Options: Default

Conditions: In all cases

V4 behavior: Call to runtime routine

V6 behavior: Call to a more efficient runtime routine

Source Example:

```
SEARCH ALL table
   AT END
     statements
   WHEN conditions
     statements
```

Performance: V6 is 50% faster than V4

Tables

Indexed tables

Statement: Accessing data items in indexed tables

Options: OPT(2)

Conditions: In all cases

V6 behavior: An efficient sequence is used to access indexed table elements by caching the offset to the start of the table in a globally available register versus having to reload this each time

Source Example:

```
1  TAB.
   5 TABENTS OCCURS 40 TIMES INDEXED BY TABIDX.
      10 TABENT1    PIC X(4) VALUE SPACES.
      10 TABENT2    PIC X(4) VALUE SPACES.

IF TABENT1 (TABIDX) NOT = TABENT2 (TABIDX)
    statements
END-IF
```

Performance: V6 is 17% faster than V4

Conditional expressions

Comparing small data items to constants

Statement: conditional expressions

Data types: *DISPLAY*, COMP-3

Options: OPT(1 | 2)

Conditions: The data item has 8 or fewer digits if zoned, or 15 or fewer digits if packed.

V4 behavior: Using in-memory instructions, modifies the sign code of the data item to a known value, then compares to a constant.

V6 behavior: Loads the value of the data item into a register, then modifies the sign code and performs the comparison in a register. This is a new optimization in V6.

Source Example:

```
01 A PIC 9(4).
```

```
If A = 0 THEN
...
```

Performance: This depends on the architecture level. On zEC12, V6 is 60% faster than V4. On z13® and z14, V6 is up to 91% faster than V4.

Chapter 3. How to tune compiler options to get the most out of V6

Enterprise COBOL V6 offers a number of new and substantially changed compiler options that can affect performance. This section highlights these options and gives recommendations on the optimal settings in order to achieve the best possible performance for your application.

Recommended compiler option set for best performance is: OPT(2), ARCH(x)

These options improve performance through:

- Maximum level of optimization - OPT(2)
- Deepest architecture exploitation – ARCH(x), where x = 7 | 8 | 9 | 10 | 11 | 12. Set the value as high as possible in accordance with the recommendations in this document and *Enterprise COBOL for z/OS Programming Guide*.

Additional settings for maximum performance applicable to some users are: STGOPT, AFP(NOVOLATILE), HGPR(NOPRESERVE)

These options improve performance through:

- Removal of unreferenced data items – STGOPT
- Omitting of saves/restores for floating point and high word registers – AFP and HGPR

Note: There are some important prerequisites for using these additional options as discussed below, and in *Compiler options* in the *Enterprise COBOL for z/OS Programming Guide*. Read and understand these options settings completely before using.

In short, these restrictions are:

- STGOPT - You cannot use STGOPT if you relies upon any of the following data items:
  - Unreferenced `LOCAL-STORAGE` and non-external `WORKING-STORAGE` level-77 and level-01 elementary data items
  - Non-external level-01 group items if none of their subordinate items are referenced
  - Unreferenced special registers
- Omitting saves/restores for high word registers - HPGR
- HGPR(NOPRESERVE) – must only be set when the caller is Enterprise COBOL, Enterprise PL/I or z/OS XL C/C++ compiler-generated code

Next we will discuss the considerations when setting these and other performance-related compiler options.

related references
Performance-related compiler options
(*Enterprise COBOL for z/OS Programming Guide*)

AFP

Default
AFP(NOVOLATILE)

Recommended
AFP(NOVOLATILE)

Reasoning

When AFP(VOLATILE) is specified, values cannot be saved in registers FP8-FP15 during calls. They must instead be saved in memory and subsequently restored. The performance impact is most significant for small programs called many times.

The use of AFP(NOVOLATILE) over AFP(VOLATILE) reduces the overhead of a program call by 10% at OPT(2). Note this was measured in an otherwise empty COBOL program to emphasize the performance cost of this option and would be less of an overall degradation in a more substantial called program.

Considerations

Specifying AFP(NOVOLATILE) requires a CICS® Transaction Server V4.1 or later.

related references

AFP (*Enterprise COBOL for z/OS Programming Guide*)

ARCH

Default

ARCH(7)

Recommended

ARCH(x) where x is the lowest level of hardware your application will have to be run on, including any disaster recovery systems.

Reasoning

Higher ARCH settings enable the compiler to exploit features of the corresponding and all earlier hardware models in order to achieve the best performance.

Note: Your application might abend if it runs on a processor with an architecture level lower than what you specified with the ARCH option.

Considerations

None besides matching to hardware level

By varying only ARCH and keeping the other options at their best recommended settings, the following performance improvements were measured over a set of IBM internal performance benchmarks:

Table 1. ARCH levels with average improvement

ARCH Levels	**Average % Improvement**
ARCH(8) vs. ARCH(7)	0.4%
ARCH(9) vs. ARCH(8)	0.3%
ARCH(10) vs. ARCH(9)	9.1%
ARCH(11) vs. ARCH(10)	0.9%
ARCH(12) vs. ARCH(11)	8.7%

When moving from ARCH(7) directly to ARCH(12), the average performance gain on these same set of benchmarks is 18.4%.

Note that only the ARCH compiler option was changed for the numbers above, and the underlying hardware was an IBM z14™ machine in all cases. This means the performance gains are strictly from compiler improvements in optimizing the COBOL applications tested.

This set of benchmarks is a mix of computation intensive and I/O intensive applications. Computation intensive applications see the largest improvement: for example, one computation intensive benchmark is reduced by 83% in execution time moving from ARCH(7) to ARCH(12). Other benchmarks spend the majority of their time performing I/O operations, and relatively little time

in compiler-generated code. The performance of these benchmarks is not significantly affected by the ARCH option.

For reference, the mapping between ARCH settings and hardware models is provided below:

Table 2. ARCH settings and hardware models

ARCH	Hardware Models
ARCH(7)	2094-xxx models (IBM System z9® EC) 2096-xxx models (IBM System z9 BC)
ARCH(8)	2097-xxx models (IBM System z10® EC) 2098-xxx models (IBM System z10 BC)
ARCH(9)	2817-xxx models (IBM zEnterprise® z196 EC) 2818-xxx models (IBM zEnterprise z114 BC)
ARCH(10)	2827-xxx models (IBM zEnterprise EC12) 2828-xxx models (IBM zEnterprise BC12)
ARCH(11)	2964-xxx models (IBM z13®) 2965-xxx models (IBM z13s®)
ARCH(12)	3906-xxx models (IBM z14™) 3907-xxx (IBM z14® ZR1) models

related references
ARCH
(*Enterprise COBOL for z/OS Programming Guide*)

ARITH

Default
ARITH(COMPAT)

Recommended
Use ARITH(EXTEND) only if the larger maximum number of digits enabled by this option is required (31 instead of 18). Otherwise, use ARITH(COMPAT) as it can result in better performance in some cases.

Reasoning

In addition to allowing larger variables to be declared, ARITH(EXTEND) also raises the maximum number of digits maintained for intermediate results. These larger intermediate results sometimes require different, slower code to be generated. Inline calculations may need to be replaced with more expensive runtime library routines.

For example, the comp-1 floating point exponentiation:

```
COMPUTE C = A ** B
```

Is 63% faster when using ARITH(COMPAT) compared to ARITH(EXTEND).

related references
ARITH (*Enterprise COBOL for z/OS Programming Guide*)

AWO

Default
NOAWO

Recommended
AWO, unless the written record is required to be updated on disk as soon as possible

Reasoning

A large reduction of EXCPs is possible by combining records written together in a block, resulting in faster file output operations, and lower CPU usage.

The AWO compiler option causes the APPLY WRITE-ONLY clause to be in effect for all physical sequential, variable-length, blocked files, even if the APPLY WRITE-ONLY clause is not specified in the program. With APPLY WRITE-ONLY in effect, the file buffer is written to the output device when there is not enough space in the buffer for the next record. Without APPLY WRITE-ONLY, the file buffer is written to the output device when there is not enough space in the buffer for the maximum size record. If the application has a large variation in the size of the records to be written, using APPLY WRITE-ONLY can result in a performance savings, since this will generally result in fewer calls to Data Management Services to handle the I/Os.

Notes:

- The APPLY WRITE-ONLY clause can be used on the physical sequential, variable-length, blocked files in the program instead of using the AWO compiler option. However, to obtain the full performance benefit, the APPLY WRITE-ONLY clause would have to be used on every physical sequential, variable-length, blocked file in the program. When used this way, the performance benefits will be the same as using the AWO compiler option.
- The AWO compiler option has no effect on a program that does not contain any physical sequential, variable-length, blocked files.

As a performance example, one test program using variable-length blocked files and AWO was 90% faster than NOAWO. This faster processing was the result of using 98% fewer EXCPs to process the writes.

related references
AWO (*Enterprise COBOL for z/OS Programming Guide*)

BLOCK0

Default
NOBLOCK0

Recommended
BLOCK0

Reasoning

Blocked I/O can reduce the number of physical I/O transfers, resulting in fewer EXCPs. The BLOCK0 compiler option changes the default for QSAM files from unblocked to blocked (as if the BLOCK CONTAINS 0 clause were specified for the files), and thus gain the benefit of system-determined blocking for output files. BLOCK0 activates an implicit BLOCK CONTAINS 0 clause for each file in the program that meets all of the following criteria:

- The FILE-CONTROL paragraph either specifies ORGANIZATION SEQUENTIAL or omits the ORGANIZATION clause.
- The FD entry does not specify RECORDING MODE U.
- The FD entry does not specify a BLOCK CONTAINS clause.

As a performance example, one test program using BLOCK0 that meets the above criteria was 90% faster than a corresponding one using NOBLOCK0, and used 98% fewer EXCPs.

related references
BLOCK0 (*Enterprise COBOL for z/OS Programming Guide*)

DATA(24) and DATA(31)

Default
DATA(31)

Recommended
DATA(31), if the program doesn't need to call and pass parameters to AMODE 24 subprograms.

Reasoning

Using DATA(31) with your RENT program will help to relieve some below the line virtual storage constraint problems. When you use DATA(31) with your RENT programs, most QSAM file buffers can be allocated above the 16MB line. When you use DATA(31) with the runtime option HEAP(,,ANYWHERE), all non-EXTERNAL WORKING-STORAGE and non-EXTERNAL FD record areas can be allocated above the 16MB line.

With DATA(24), the WORKING-STORAGE and FD record areas will be allocated below the 16 MB line.

Notes:

- For NORENT programs, the RMODE option determines where non-EXTERNAL data is allocated.
- See QSAM buffers for additional information on QSAM file buffers.
- See ALL31 for information on where EXTERNAL data is allocated.
- LOCAL-STORAGE data is not affected by the DATA option. The STACK runtime option and the AMODE of the program determine where LOCAL-STORAGE is allocated.

Note that while it is not expected to impact the performance of the application, it does affect where the program's data is located.

related references
DATA (*Enterprise COBOL for z/OS Programming Guide*)

DYNAM

Default
NODYNAM

Considerations

The DYNAM compiler option specifies that all subprograms invoked through the CALL literal statement will be loaded dynamically at run time. This allows you to share common subprograms among several different applications, allowing for easier maintenance of these subprograms since the application will not have to be relinked if the subprogram is changed. DYNAM also allows you to control the use of virtual storage by giving you the ability to use a CANCEL statement to free the virtual storage used by a subprogram when the subprogram is no longer needed. However, when using the DYNAM option, you pay a performance penalty since the call must go through a library routine, whereas with the NODYNAM option, the call goes directly to the subprogram. Hence, the path length is longer with DYNAM than with NODYNAM.

As a performance example of using CALL literal in a CALL intensive program (measuring CALL overhead only), the overhead associated with the CALL using DYNAM was around 100% slower than NODYNAM. The result is affected by the number of calls to the same program. A larger number of calls tend to better amortize the overhead cost of loading the subprogram.

For additional considerations using call literal and call identifier, see “Using CALLs” on page 42.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

related references
DYNAM (*Enterprise COBOL for z/OS Programming Guide*)

FASTSRT

Default
NOFASTSRT

Recommended
FASTSRT, if COBOL file error handling semantics is not needed during the sort processing.

Reasoning

For eligible sorts, the FASTSRT compiler option specifies that the SORT product will handle all of the I/O and that COBOL does not need to do it. This eliminates all of the overhead of returning control to COBOL after each record is read in, or after processing each record that COBOL returns to SORT. The use of FASTSRT is recommended when direct access devices are used for the sort work files, since the compiler will then determine which sorts are eligible for this option and generate the proper code. If the sort is not eligible for this option, the compiler will still generate the same code as if the NOFASTSRT option were in effect. A list of requirements for using the FASTSRT option is in the COBOL programming guide.

As a performance example, one test program that processed 100,000 records was 45% faster when using FASTSRT compared to using NOFASTSRT and used 4,000 fewer EXCPs.

related references
FASTSRT (*Enterprise COBOL for z/OS Programming Guide*)

HGPR

Default
HGPR(PRESERVE)

Recommended
HGPR(NOPRESERVE)

Reasoning

Better performance as no code has to be generated to save on entry and restore on exit the high halves of the 64-bit GPRs. Using the recommended HGPR setting is particularly important to improve the performance of relatively small COBOL programs that are entered many times.

When HGPR(PRESERVE) is specified, the compiler cannot rely on the high halves of general purpose registers (GPRs) being preserved during calls. Instead, they must be saved in memory and subsequently restored. The performance impact is most weakened for small programs that are called many times.

The use of HGPR(NOPRESERVE) over HGPR(PRESERVE) reduces the overhead of a program call by 6% at OPT(2). Note this was measured in an otherwise empty COBOL program to emphasize the performance cost of this option and would be less of an overall degradation in a more substantial callee program.

Considerations
The PRESERVE suboption is necessary only if the caller of the program is not Enterprise COBOL, Enterprise PL/I, or z/OS XL C/C++ compiler-generated code.

related references
HGPR (*Enterprise COBOL for z/OS Programming Guide*)

INLINE

Default
INLINE

Recommended
INLINE

Reasoning

When the INLINE option is specified, the compiler may choose to replace the PERFORM of a paragraph or section with a copy of that paragraph or section's code. By inserting that code at the location of the PERFORM, the compiler saves the overhead of branching logic to and from the procedure. Inlining also allows the compiler to perform further optimizations on the inlined code. For example, a data item used inside the inlined code may have a known constant value at that PERFORM, allowing the compiler to simplify expressions.

When NOINLINE is specified, the compiler will not create any inlined duplicate code.

Considerations

The >>INLINE OFF and >>INLINE ON directives enable more fine-grained control over inlining. This can help reduce program size and improve instruction cache performance in cases where PERFORMs are rarely executed (for example, in error handling code).

related references
INLINE (*Enterprise COBOL for z/OS Programming Guide*)

INVDATA

Default
NOINVDATA

Recommended

NOINVDATA

Because most users have valid data in their `USAGE DISPLAY` and `USAGE PACKED-DECIMAL` data items, use `NOINVDATA` to improve the performance of your application. Even if you find that your programs are processing invalid data at run time with the `NUMCHECK` compiler option, you should change your programs to avoid processing invalid data and use `NOINVDATA`.

Note: The goal of the `INVDATA` option is to provide a behavior that is as compatible as possible with the behavior of programs compiled with COBOL V4 or earlier versions in cases of invalid numeric data. When discrepancies are found, this option will be updated in favor of making the behavior more closely match the behavior of COBOL V4 or earlier versions.

When the `INVDATA` option is in effect, the compiler will avoid performing known optimizations that might produce a different result than COBOL V4 or earlier versions when a zoned decimal or packed decimal data item has invalid digits or an invalid sign code, or when a zoned decimal data item has invalid zone bits.

The following table provides a quick reference on how to set the `INVDATA` and `NUMPROC` options when migrating to COBOL V6.2 or later versions from earlier versions of COBOL, depending on the default value of the `NUMPROC` option that was used in the earlier version of COBOL and whether or not you have invalid data.

Table 3. Setting INVDATA and NUMPROC options when migrating from earlier COBOL versions

COBOL versions	Invalid data present?	NUMPROC/ZONEDATA used in COBOL V6.1 or earlier versions	INVDATA and NUMPROC settings in COBOL V6.2 or later versions
Pre-COBOL V5	No	`NUMPROC(MIG)`	`NOINVDATA,NUMPROC(NOPFD)`
Pre-COBOL V5	No	`NUMPROC(NOPFD)`	`NOINVDATA,NUMPROC(NOPFD)`
Pre-COBOL V5	No	`NUMPROC(PFD)`	`NOINVDATA,NUMPROC(PFD)`

Table 3. Setting INVDATA and NUMPROC options when migrating from earlier COBOL versions (continued)

COBOL versions	Invalid data present?	NUMPROC/ZONEDATA used in COBOL V6.1 or earlier versions	INVDATA and NUMPROC settings in COBOL V6.2 or later versions
Pre-COBOL V5	Yes	`NUMPROC(MIG)`	`INVDATA(FORCENUMCMP, NOCLEANSIGN),NUMPROC(NOPFD)`
Pre-COBOL V5	Yes	`NUMPROC(NOPFD)`	`INVDATA(NOFORCENUMCMP,CLEANSIGN),NUMPROC(NOPFD)` or `INVDATA,NUMPROC(NOPFD)`
Pre-COBOL V5	Yes	`NUMPROC(PFD)`	`INVDATA(NOFORCENUMCMP,CLEANSIGN),NUMPROC(PFD)` or `INVDATA,NUMPROC(PFD)`
COBOL V5 or later	No	`ZONEDATA(PFD)`	`NOINVDATA`
COBOLV5 or later	Yes	`ZONEDATA(NOPFD)`	`INVDATA(NOFORCENUMCMP,CLEANSIGN)` or simply `INVDATA`
COBOL V5 or later	Yes	`ZONEDATA(MIG)`	`INVDATA(FORCENUMCMP, CLEANSIGN)`[1]

1. `INVDATA(FORCENUMCMP,NOCLEANSIGN)` is a closer representation of the pre-COBOL V5 `NUMPROC(MIG)` behavior than `INVDATA(FORCENUMCMP,CLEANSIGN)` when invalid data is present. If you are not satisfied with the behavior of `ZONEDATA(MIG)` in COBOL V5 or later versions when invalid data is present, then consider using `INVDATA(FORCENUMCMP,NOCLEANSIGN)` to more closely mimic the pre-COBOL V5 `NUMPROC(MIG)` behavior when invalid data is present.

For details about the INVDATA option and how the compiler behaves when the sign code, digits, or zone bits are invalid, see *INVDATA* in the *Enterprise COBOL for z/OS Programming Guide*.

Reasoning

- When the NOINVDATA option is in effect, the compiler assumes that the data in `USAGE DISPLAY` and `PACKED-DECIMAL` data items are valid, and generates the most efficient code possible to make numeric comparisons. For example, the compiler might generate a string comparison to avoid numeric conversion.
- When the INVDATA(FORCENUMCMP) option is in effect, the compiler must generate additional instructions to do numeric comparisons that ignore the zone bits of each digit in zoned decimal data items. For example, a zoned decimal value might be converted to packed-decimal with a PACK instruction before the comparison.
- When the INVDATA(NOFORCENUMCMP) option is in effect, the V6 compiler must generate a sequence that treats the invalid zone bits, the invalid sign code and the invalid digits in the same way as the V4 compiler, even when that sequence is less efficient than another possible sequence. The following cases are considered:
  - In the cases where COBOL V4 or earlier versions considered the zone bits, the compiler generates an alphanumeric comparison which will also consider the zone bits of each digit in zoned decimal data items. The zoned decimal value remains as zoned decimal.

– In the cases where COBOL V4 or earlier versions ignored the zone bits of each digit in zoned decimal data items. The zoned decimal value is converted to packed-decimal with a PACK instruction before the comparison.

Source Example

```
01 A PIC S9(5)V9(2).
01 B PIC S9(7)V9(2).
COMPUTE B = A * 100
```

In this example, the multiplication is 32% faster with NOINVDATA than INVDATA. With NOINVDATA, the compiler can use a shift instruction instead of a multiplication. With INVDATA, it must perform the more expensive multiplication.

Related references
INVDATA (*Enterprise COBOL for z/OS Programming Guide*)

MAXPCF

Default

MAXPCF(100000)

Recommended

MAXPCF(0)

Reasoning

MAXPCF can be specified to automatically reduce the amount of optimization for large and complex programs that may require excessive compilation time or excessive storage requirements.

The MAXPCF option is intended to allow large programs to compile successfully but at the cost of reduced optimization. However, if possible, it is recommended to restructure your large applications into smaller separate programs.

A number of new "global" optimizations have been added to the V5 compiler release. These optimizations are termed "global" as they attempt to find synergies and improve performance across an entire program instead of just "locally" within a statement or a linear set of statements in a section. Because these global optimizations must analyze the statements and data items of the entire program, they sometimes require significant amounts of storage and time.

For this reason, and to generally benefit software maintenance activities, it is strongly recommended to break large programs into smaller separate programs linked together by static calls (to minimize overhead versus using dynamic calls). These smaller programs will have a much better chance of not requiring a downgrade of optimization by the MAXPCF option. This will also likely result in faster compile times requiring less storage, and the final compiled and linked application will have been subject to the full suite of optimizations available in the V6 compiler.

Considerations

Specifying MAXPCF(0) may increase compilation time.

related references
MAXPCF (*Enterprise COBOL for z/OS Programming Guide*)

NUMCHECK

Default
NONUMCHECK

Recommended
For best performance, NONUMCHECK is recommended. Specifying NUMCHECK will cause IS NUMERIC class tests to be generated whenever a numeric data item is used as a sender.

Reasoning

The extra checks inserted by NUMCHECK can cause significant performance degradations for programs that use zoned decimal data items as sending data items. It is faster to manually insert IS NUMERIC tests at places in your program where data is read into your program, instead of using NUMCHECK that will enable checks for all cases, including the performance-critical parts of your program.

For example, the zoned decimal move:

```
01 Z1 PIC 9(5).
01 Z2 PIC 9(5).
MOVE Z1 TO Z2.
```

is 52% faster when using NONUMCHECK compared to NUMCHECK (MSG) or NUMCHECK (ABD).

Considerations

In addition to the runtime performance impact, use of SSRANGE can also significantly increase compilation time.

Note: NUMCHECK(ZON) was previously known as ZONECHECK.

related references

NUMCHECK (*Enterprise COBOL for z/OS Programming Guide*)

NUMPROC

Default

NUMPROC(NOPFD)

Recommended

When your numeric data exactly conforms to the IBM system standards as detailed in *NUMPROC* in the *Enterprise COBOL for z/OS Programming Guide*, use NUMPROC(PFD) to improve the performance of your application.

Note: Changing the NUMPROC option can lead to different behaviour if you have invalid data. Make sure that you thoroughly test before using NUMPROC(PFD) to avoid unpredictable results. Use the NUMCHECK(ZON,PAC) option to have the compiler generate implicit numeric class tests to validate your numeric data. This can help you decide whether you can use NUMPROC(PFD).

For details about the NUMCHECK option, see *NUMCHECK* in the *Enterprise COBOL for z/OS Programming Guide*.

Reasoning

NUMPROC(PFD) improves performance as the compiler no longer has to generate code to correct input sign configurations. This is particularly important when your application contains unsigned internal decimal and zoned decimal data, as this type of data requires correction before use in any arithmetic or compare statements, in addition to also correcting after certain arithmetic, move and compare statements.

A benchmark that contains many types of arithmetic improves by 1.3% when using NUMPROC(PFD) compared to NUMPROC(NOPFD)

related references

NUMPROC (*Enterprise COBOL for z/OS Programming Guide*)

OPTIMIZE

Default

OPT(0)

Recommended

OPT(2)

Reasoning

Maximum level of optimization generally results in the fastest performing code to be generated by the compiler.

Considerations

OPT(2) compiles generally use more memory and take longer to complete compared to using OPT(1) or OPT(0).

Compile-time data gathered from a set of benchmarks show that on average OPT(1) takes 1.5 times longer than OPT(0), and OPT(2) takes 1.8 times longer than OPT(0) (comparing CPU time). For very large test cases, the compile-time trade off can be worse than the average.

In addition, debuggability can be reduced as compiler optimizations and dead code removal are more advanced at this setting.

The possible settings for the OPTIMIZE option changed between V4 and V5. V6 continues to use the new V5 settings. The meaning of those settings has not changed between V5 and V6.

Note that unreferenced level 01 and level 77 items are no longer deleted with this highest OPT setting as was the case in V4 with OPT(FULL). This means programs that could not use OPT(FULL) previously can specify OPT(2). See “STGOPT” on page 26 for more information.

Although both V4 and V6 offer three levels of OPT specifications, the names and more importantly the underlying optimizations enabled have changed.

For example, an important difference between V4 and V6 is that the highest setting in V4 of OPT(FULL) was the suite of OPT(STD) optimizations plus the removal of unreferenced data items and the corresponding code to initialize their VALUE clauses.

In contrast, the highest setting in V6 is OPT(2) and this contains the suite of OPT(1) optimizations plus additional optimizations to improve performance, such as globally propagating values, sign state information and better register allocation for accessing indexed tables.

As detailed in *OPTIMIZE* in the *Enterprise COBOL for z/OS Programming Guide*, the table of "Mapping of deprecated options to new options", the V4 OPT settings are currently tolerated, but none of the V4 settings map to OPT(2). For example, OPT(FULL) specified with V6 is mapped to OPT(1) and STGOPT.

related references

OPTIMIZE (*Enterprise COBOL for z/OS Programming Guide*)

SSRANGE

Default

NOSSRANGE

Recommended

For best performance, NOSSRANGE is recommended. Specifying SSRANGE will cause extra code to be generated to detect out of range storage references.

Reasoning

The extra checks enabled by SSRANGE can cause significant performance degradations for programs with index, subscript and reference modification expressions in performance sensitive areas of your program. If only a few places in your program require the extra range checking, then it might be faster to code your own checks instead of using SSRANGE which will enable checks for all cases.

Note that in COBOL 6, there is no longer a runtime option to disable the compiled-in checks. So specifying SSRANGE will always result in the range checking code to be used at runtime. A benchmark that makes moderate use of subscripted references to tables slows down by 18% when SSRANGE is specified.

There is no performance difference between SSRANGE(ZLEN) and SSRANGE(NOZLEN).

Considerations

In addition to the runtime performance impact, use of SSRANGE can also significantly increase compilation time.

related references
SSRANGE (*Enterprise COBOL for z/OS Programming Guide*)

STGOPT

Default
NOSTGOPT

Recommended
STGOPT

Reasoning
This is a new option introduced in V5 that is now orthogonal to OPT. In V4, the STGOPT behavior to remove unreferenced data items and the corresponding code to initialize their VALUE clauses is implied when going from OPT(STD) to OPT(FULL). Since V5, that behavior is now specified independently. Over a set of benchmark programs, the use of STGOPT results in an average 2.8% reduction in the size of the object file at OPT(2), and a maximum reduction of 11.8%.

Considerations
The same considerations that applied in V4 to specifying OPT(FULL) should be used in deciding to use STGOPT in V6. That is, you cannot use neither OPT(FULL) nor STGOPT if you relies upon any of the following data items:

- Unreferenced `LOCAL-STORAGE` and non-external `WORKING-STORAGE` level-77 and level-01 elementary data items
- Non-external level-01 group items if none of their subordinate items are referenced
- Unreferenced special registers

Note: The STGOPT option is ignored for data items that have the VOLATILE clause.

related references
STGOPT (*Enterprise COBOL for z/OS Programming Guide*)

TEST

See *TEST* in the *Enterprise COBOL for z/OS Programming Guide* for a full discussion of the TEST option and suboptions including a discussion of performance versus debugging capability tradeoffs.

To summarize the performance tradeoffs of programs compiled with TEST options:

- NOTEST performs better than TEST(NOEJPD)
- TEST(NOEJPD) performs significantly better than TEST(EJPD)

TEST(EJPD) enables the JUMPTO and GOTO commands and therefore puts severe restrictions on the amount optimization performed by the compiler. The EJPD suboption limits the compiler to in-statement optimizations to allow the JUMPTO and GOTO commands to work properly.

Note: If you specify `TEST(NOEJPD)` and a non-zero `OPTIMIZE` level: The `JUMPTO` and `GOTO` commands are not enabled, but you can use `JUMPTO` and `GOTO` if you use the `SET WARNING OFF` command. In this scenario, `JUMPTO` and `GOTO` will have unpredictable results.

TEST(NOEJPD) also restricts the optimizer, but much less so than TEST(EJPD). The NOEJPD suboption allows the viewing of data items at statement boundaries and this restricts the optimizer in removing some dead code and dead stores.

The table below shows average execution time performance numbers over a set of IBM internal performance benchmarks.

The numbers were produced at OPT(1) and OPT(2), using different TEST suboptions. The percentages show the performance degradations of TEST(NOEJPD) or TEST(EJPD) over NOTEST.

Table 4. Performance degradations of TEST(NOEJPD) or TEST(EJPD) over NOTEST

OPT level	TEST(NOEJPD) % degradation versus NOTEST	TEST(EJPD) % degradation versus NOTEST
OPT(1)	5.8%	20.9%
OPT(2)	11.1%	28.5%

As expected, this demonstrates the much larger impact on performance of TEST(EJPD) versus TEST(NOEJPD).

related references
TEST (*Enterprise COBOL for z/OS Programming Guide*)

THREAD

Default
NOTHREAD

Recommended
NOTHREAD

Reasoning

The THREAD option requires additional locking in the generated code and the COBOL runtime library, which can impact performance. This is unnecessary if the program is not running in a multi-threaded environment.

This applies not just to Enterprise COBOL V6, but also applies to previous Enterprise COBOL compilers.

The THREAD option indicates that a COBOL program is to be enabled for execution in an environment that has multiple POSIX threads or PL/I tasks. In order to do so, the compiler inserts locks in various places in the generated code to protect the execution. This can impact performance of a THREAD compiled program in comparison with a corresponding NOTHREAD program.

It is recommended that the NOTHREAD option is used unless the program requires it. The compiler default is NOTHREAD.

One example where the compiler needs to insert locks to protect is I/O. When the THREAD option is used, all I/O verbs (OPEN, READ, WRITE, REWRITE, CLOSE, etc) are protected by locks. In a measurement, we observed a performance degradation of 10% due to the THREAD option.

related references
THREAD (*Enterprise COBOL for z/OS Programming Guide*)

TRUNC

Default
TRUNC(STD)

Recommended
The recommended option for best performance continues to be TRUNC(OPT), as this allows the compiler the most freedom in determining the most efficient code to generate. For additional information on determining which TRUNC option to specify, see *TRUNC* in the *Enterprise COBOL for z/OS Programming Guide*.

Note: Changing the TRUNC option can lead to different behaviour if you have invalid data. Use the TRUNC(OPT) option only if you are sure that the data being moved into the binary areas will

not have a value with larger precision than that defined by the PICTURE clause for the binary item. Otherwise, unpredictable results could occur. To be sure, perform thorough testing using the NUMCHECK(BIN(TRUNCBIN)) option, which causes the compiler to test if binary data items contents are bigger than the PICTURE clause. This can help you decide whether you can use TRUNC(OPT).

For details about the NUMCHECK option, see *NUMCHECK* in the *Enterprise COBOL for z/OS Programming Guide*.

Reasoning

The cost of using TRUNC(STD) has been improved compared to V4, as the divide instruction used to truncate the result back to the number of digits in the PICTURE clause of the BINARY receiving data item is only conditionally executed in V6. The compiler inserts a runtime check for overflow and will branch around the divide if no truncation is required.

However, better performance is still possible when using TRUNC(OPT) as no runtime overflow checks or divide instructions are required at all.

TRUNC(BIN) will often result in poorer performance, and is usually the slowest of the three TRUNC suboptions. Although no divides (conditional or otherwise) are required in order to truncate results, the full 2, 4 or 8 byte value is considered significant and therefore intermediate results grow that much more quickly and require conversions to larger or more complex data types.

For example, when adding two BINARY PIC 9(10) values with TRUNC(STD) or TRUNC(OPT), the maximum result size is 11 digits. No overflow is possible. The addition can be performed using binary arithmetic. When performing the same addition with TRUNC(BIN), each operand can have up to 18 digits, and the maximum result size is 19 digits. This may overflow. Therefore, the operands must be converted to packed decimal before performing the addition. This is slower.

Similarly, when multiplying two BINARY PIC 9(10) values with TRUNC(STD) or TRUNC(BIN), the maximum result size is 20 digits. This is too large for a binary operation, but not too large for packed decimal arithmetic. When performing the same multiplication with TRUNC(BIN), each operand can have up to 18 digits, and the maximum result size is 36 digits. Because hardware support for packed decimal arithmetic is limited to 31 digits, this multiplication requires an expensive runtime call.

Specifically, adding two BINARY PIC 9(10) items together is 10% faster using TRUNC(OPT) than TRUNC(STD), and 97% faster using TRUNC(OPT) than TRUNC(BIN).

In one program with a significant amount of binary arithmetic setting TRUNC(BIN) results in a 76% slowdown compared to TRUNC(STD). This performance difference is due to the runtime library calls required for the larger intermediate result sizes.

See “BINARY (COMP or COMP-4)” on page 51 for a more detailed discussion and study of BINARY data and interaction with TRUNC suboptions.

Related references
TRUNC (*Enterprise COBOL for z/OS Programming Guide*)

related references
TRUNC (*Enterprise COBOL for z/OS Programming Guide*)

Program residence and storage considerations

Compiler option

The following compiler options can affect where the program resides (above/below the 16 MB line), which in turn can affect the location of WORKING-STORAGE section, and I/O file buffers and record areas.

RENT or NORENT

Using the RENT compiler option causes the compiler to generate some additional code to ensure that the program is reentrant. Reentrant programs can be placed in shared storage like the Link Pack Area (LPA) or

the Extended Link Pack Area (ELPA). Also, the RENT option will allow the program to run above the 16 MB line. Producing reentrant code may increase the execution time path length slightly.

Note: The RMODE(ANY) option can be used to run NORENT programs above the 16 MB line.

Performance considerations using RENT: On the average, RENT was equivalent to NORENT.

For details, see *RENT* in the *Enterprise COBOL for z/OS Programming Guide*.

RMODE - AUTO, 24, or ANY

The RMODE compiler option determines the RMODE setting for the COBOL program. When using RMODE(AUTO), the RMODE setting depends on the use of RENT or NORENT. For RENT, the program will have RMODE ANY. For NORENT, the program will have RMODE 24. When using RMODE(24), the program will always have RMODE 24. When using RMODE(ANY), the program will always have RMODE ANY.

Note: When using NORENT, the RMODE option controls where the WORKING-STORAGE will reside. With RMODE(24), the WORKING-STORAGE will be below the 16 MB line. With RMODE(ANY), the WORKING-STORAGE can be above the 16 MB line.

While it is not expected to impact the performance of the application, it can affect where the program and its WORKING-STORAGE are located.

For details, see *RMODE* in the *Enterprise COBOL for z/OS Programming Guide*.

Location of Storage

WORKING-STORAGE

COBOL WORKING-STORAGE is allocated from the Language Environment heap storage when the program is compiled with the RENT option.

LOCAL-STORAGE

COBOL LOCAL-STORAGE is always allocated from the Language Environment stack storage. It is affected by the LE STACK runtime option.

EXTERNAL variables

External variables in an Enterprise COBOL program are always allocated from the Language Environment heap storage.

QSAM buffers

QSAM buffers can be allocated above the 16 MB line if all of the following are true:

- The programs are compiled with VS COBOL II Release 3.0 or higher, COBOL/370 Release 1.0 or higher, IBM COBOL for MVS™ & VM Release 2.0 or higher, IBM COBOL for OS/390® & VM, or IBM Enterprise COBOL
- The programs are compiled with RENT and DATA(31) or compiled with NORENT and RMODE(ANY)
- The program is executing in AMODE 31
- The ALL31(ON) and HEAP(,,ANYWHERE) runtime options are used (for EXTERNAL files)
- The file is not allocated to a TSO terminal
- The file is not spanned external, spanned with a SAME RECORD clause, or spanned opened as I-O and updated with REWRITE

For details, see *Allocation of buffers for QSAM files* in the *Enterprise COBOL for z/OS Programming Guide*.

VSAM buffers

VSAM buffers can be allocated above the 16 MB line if the programs are compiled with VS COBOL II Release 3.0 or higher, COBOL/370 Release 1.0 or higher, IBM COBOL for MVS & VM Release 2.0 or higher, IBM COBOL for OS/390 & VM, or IBM Enterprise COBOL.

Chapter 4. Runtime options that affect runtime performance

Selecting the proper runtime options affects the performance of a COBOL application.

Therefore, it is important for the system programmer responsible for installing and setting up the LE environment to work with the application programmers so that the proper runtime options are set up correctly for your installation. It is also important to understand these options so that you can set the appropriate options for specific programs, applications, and regions that require fast performance. The individual LE runtime options can be set using any of the supported methods for setting that individual option. Below examines some of the options that can help to improve the performance of the individual application, as well as the overall LE runtime environment.

Note: In the following option description, if an option setting is different between CICS and non-CICS, the setting will be qualified by text in parentheses. Otherwise the same setting applies to both CICS and Non-CICS.

related references
Using runtime options (*z/OS Language Environment Programming Guide*)
Summary of Language Environment runtime options
(*z/OS Language Environment Programming Reference*)
Using the Language Environment runtime options
(*z/OS Language Environment Programming Reference*)
Language Environment runtime options
(*z/OS Language Environment Customization*)

AIXBLD

Default
OFF (Non-CICS); N/A (CICS)

Recommended
OFF (Non-CICS); N/A (CICS)

Considerations
The AIXBLD option allows alternate indexes to be built at run time. However, this may adversely affect the runtime performance of the application. It is much more efficient to use Access Method Services to build the alternate indexes before running the COBOL application than using the AIXBLD runtime option. Note that AIXBLD is not supported when VSAM data sets are accessed in RLS mode.

related references
AIXBLD (COBOL only) (*z/OS Language Environment Programming Reference*)
AIXBLD (COBOL only) (*z/OS Language Environment Customization*)

ALL31

Default
ON

Recommended
ON, unless there are AMODE(24) routines in the application

Considerations
The ALL31 option allows LE to take advantage of the knowledge that there are no AMODE(24) routines in the application.

ALL31(ON) specifies that the entire application will run in AMODE(31). This can help to improve the performance for an all AMODE(31) application because LE can minimize the amount of mode

switching across calls to common runtime library routines. Additionally, using ALL31(ON) will help to relieve some below the line virtual storage constraint problems, since less below the line storage is used.

When using ALL31(ON), all EXTERNAL WORKING-STORAGE and EXTERNAL FD records areas can be allocated above the 16MB line if you also use the HEAP(,,ANYWHERE) runtime option and compile the program with either the DATA(31) and RENT compiler options or with the RMODE(ANY) and NORENT compiler options. Note that when using ALL31(OFF), you must also use STACK(,,BELOW).

Notes:

- Beginning with LE for z/OS Release 1.2, the runtime defaults have changed to ALL31(ON),STACK(,,ANY). LE for OS/390 Release 2.10 and earlier runtime defaults were ALL31(OFF),STACK(,,BELOW).
- ALL31(OFF) is required for all OS/VS COBOL programs that are not running under CICS, all VS COBOL II NORES programs, and all other AMODE(24) programs.

As a performance example (measuring CALL overhead only), a test program using ALL31(ON) was equivalent to ALL31(OFF).

Note: This test measured only the overhead of the CALL for a RENT program (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms will have different results, depending on the number of calls that are made to LE common runtime routines.

related references
ALL31 (*z/OS Language Environment Programming Reference*)
ALL31 (*z/OS Language Environment Customization*)

CBLPSHPOP

Default
ON

Recommended
N/A (Non-CICS); ON (CICS), if compatible behavior with VS COBOL II is required in the EXEC CICS condition handling commands. If compatible behavior with VS COBOL II is not required, or if the program does not use any of the EXEC CICS condition handling commands, the OFF setting is recommended

Considerations
The CBLPSHPOP option controls whether CICS PUSH HANDLE and CICS POP HANDLE commands are issued when a COBOL subroutine is called.

This option only applies to the CICS environment. The CBLPSHPOP option is used to avoid compatibility problems when calling COBOL subroutines that contain CICS CONDITION, AID, or ABEND condition handling commands.

- When CBLPSHPOP is OFF and you want to handle these CICS conditions in your COBOL subprogram, you will need to issue your own CICS PUSH HANDLE before calling the COBOL subprogram and CICS POP HANDLE upon return. Otherwise, the COBOL subroutine will inherit the caller's settings and upon return, the caller will inherit any settings that were made in the subprogram. This behavior is different from that of VS COBOL II.
- When CBLPSHPOP is ON, you will receive the same behavior as with the VS COBOL II run time when using CICS condition handling commands. However, the performance of calls will be impacted.

For performance considerations using CBLPSHPOP, see “CICS” on page 46.

related tasks
Developing COBOL programs for CICS
(*Enterprise COBOL for z/OS Programming Guide*)

related references
Using the CBLPSHPOP runtime option under CICS

(*z/OS Language Environment Programming Guide*)
CBLPSHPOP (COBOL only) (*z/OS Language Environment Programming Reference*)
CBLPSHPOP (COBOL only) (*z/OS Language Environment Customization*)

CHECK

The CHECK runtime option is ignored for applications compiled with Enterprise COBOL V6.

If the compile time option SSRANGE is specified, range checks are generated by the compiler and checks are always executed at run time. The compiled-in range checks cannot be disabled.

related references
CHECK (COBOL only) (*z/OS Language Environment Programming Reference*)
CHECK (COBOL only) (*z/OS Language Environment Customization*)

DEBUG

Default
OFF

Recommended
OFF

Considerations
The DEBUG option activates the COBOL batch debugging features specified by the USE FOR DEBUGGING declarative. This might add some additional overhead to process the debugging statements. This option has an effect only on a program that has the USE FOR DEBUGGING declarative.

Performance considerations using DEBUG:

- When not using the USE FOR DEBUGGING declarative, on the average, DEBUG was equivalent to NODEBUG.
- When using the USE FOR DEBUGGING declarative, a test program measured was 900% slower when using DEBUG compared to using NODEBUG.

Note: The program in this test had WITH DEBUGGING MODE clause on the SOURCE-COMPUTER paragraph, and contained a USE FOR DEBUGGING ON a paragraph name in the procedure division. This paragraph is empty (that is containing just an EXIT statement), and is performed many times in a loop. The paragraph in the declarative section is also empty (just an EXIT statement). The purpose is to give an indication on the overhead due to transferring of control to the USE FOR DEBUGGING declarative.

related references
DEBUG (COBOL only) (*z/OS Language Environment Programming Reference*)
DEBUG (COBOL only) (*z/OS Language Environment Customization*)

INTERRUPT

Default
OFF (Non-CICS); N/A (CICS)

Recommended
OFF (Non-CICS); N/A (CICS)

Considerations
The INTERRUPT option causes attention interrupts to be recognized by Language Environment. When you cause an interrupt, Language Environment can give control to your application or to Debug Tool.

Performance considerations using INTERRUPT: On the average, INTERRUPT(ON) is 1% slower than INTERRUPT(OFF), with a range of equivalent to 20% slower.

related references
INTERRUPT (*z/OS Language Environment Programming Reference*)
INTERRUPT (*z/OS Language Environment Customization*)

RPTOPTS

Default
OFF

Recommended
OFF

Considerations
The RPTOPTS option allows you to get a report of the runtime options that were in use during the execution of an application. This report is produced after the application has terminated. Thus, if the application abends, the report may not be generated. Generating the report can result in some additional overhead. Specifying RPTOPTS(OFF) will eliminate this overhead.

Performance considerations using RPTOPTS: On the average, RPTOPTS(ON) was equivalent to RPTOPTS(OFF).

Note: Although the average for a single batch program shows equivalent performance for RPTOPTS(ON), you may experience some degradation in a transaction environment (for example, CICS) where main programs are repeatedly invoked.

related references
RPTOPTS (*z/OS Language Environment Programming Reference*)
RPTOPTS (*z/OS Language Environment Customization*)

RPTSTG

Default
OFF

Recommended
OFF

Considerations
The RPTSTG option allows you to get a report on the storage that was used by an application. This report is produced after the application has terminated. Thus, if the application abends, the report may not be generated. The data from this report can help you fine tune the storage parameters for the application, reducing the number of times that the LE storage manager must make system requests to acquire or free storage.

Collecting the data and generating the report can result in some additional overhead. Specifying RPTSTG(OFF) will eliminate this overhead.

Performance considerations using RPTSTG: The degradation in a call intensive program was measured to be more than 200%.

Note: The program did nothing except repeatedly calling a number of subprograms, which were empty (that is, containing only a GOBACK statement).

related references
RPTSTG (*z/OS Language Environment Programming Reference*)
RPTSTG (*z/OS Language Environment Customization*)

RTEREUS

Default
OFF (Non-CICS); N/A (CICS)

Recommended
OFF (Non-CICS); N/A (CICS)

Considerations
The RTEREUS option causes the LE runtime environment to be initialized for reusability when the first COBOL program is invoked.

The LE runtime environment remains initialized (all COBOL programs and their work areas are kept in storage) in addition to keeping the library routines initialized and in storage. This means that, for subsequent invocations of COBOL programs, most of the runtime environment initialization will be bypassed. Most of the runtime termination will also be bypassed, unless a STOP RUN is executed or unless an explicit call to terminate the environment is made (Note: using STOP RUN results in control being returned to the caller of the routine that invoked the first COBOL program, terminating the reusable runtime environment).

Because of the effect that the STOP RUN statement has on the runtime environment, you should change all STOP RUN statements to GOBACK statements in order to get the benefit of RTEREUS. The most noticeable impact will be on the performance of a non-COBOL driver repeatedly calling a COBOL subprogram (for example, a non-LE-conforming assembler driver that repeatedly calls COBOL applications). The RTEREUS option helps in this case.

However, using the RTEREUS option does affect the semantics of the COBOL application: each COBOL program will now be considered to be a subprogram and will be entered in its last-used state on subsequent invocations (if you want the program to be entered in its initial state, you can use the INITIAL clause on the PROGRAM-ID statement). This means that storage that is acquired during the execution of the application will not be freed. Since the storage is not freed, RTEREUS cannot be used with SVC LINK since after return from the SVC LINK, the program LINKed to will be deleted by the operating system, but the COBOL control blocks will still be initialized and in storage. Therefore, RTEREUS may not be applicable to all environments.

Performance considerations using RTEREUS (measuring CALL overhead only): One testcase (a non-LE-conforming Assembler calling COBOL) using RTEREUS was 99% faster than using NORTEREUS.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

related topics
Upgrading applications that use an assembler driver
(*Enterprise COBOL for z/OS Migration Guide*)

related references
RTEREUS (COBOL only) (*z/OS Language Environment Programming Reference*)
RTEREUS (COBOL only) (*z/OS Language Environment Customization*)
COBOL and Language Environment runtime options comparison
(*z/OS Language Environment Runtime Application Migration Guide*)

STORAGE

Default
NONE,NONE,NONE,0K

Recommended
NONE,NONE,NONE,0K

Considerations
The STORAGE option specifies the heap allocations or stack storage.

The first parameter of this option initializes all heap allocations, including all external data records acquired by a program, to the specified value when the storage for the external data is allocated. This also includes the WORKING-STORAGE acquired by a RENT program (see note below), unless a VALUE clause is used on the data item, when the program is first called or, for dynamic calls, when the program is canceled and then called again. In any case, storage is not initialized on subsequent calls

to the program. This can result in some overhead at run time depending on the number of external data records in the program and the size of the WORKING-STORAGE section.

The WORKING-STORAGE is affected by the STORAGE option in the following categories of RENT programs:

- When the program runs in CICS environment
- When the program is compiled with Enterprise COBOL V4.2 or earlier
- When the program is compiled with Enterprise COBOL V6.1 or later
- When the program object (where the program resides) contains only programs compiled with COBOL V5.1.1 or later, or compiled with COBOL V4.2 or earlier compilers; (i.e. there is no Language Environment interlanguage calls within the program object)
- When the primary entry point of the program object is a program compiled with Enterprise COBOL V5.1.1 or later; (i.e. having LE interlanguage calls within the program object is allowed)

Note: If you used the WSCLEAR option with VS COBOL II, STORAGE(00,NONE,NONE) is the equivalent option with Language Environment.

The second parameter of this option initializes all heap storage when it is freed.

The third parameter of this option initializes all DSA (stack) storage when it is allocated. The amount of overhead depends on the number of routines called (subroutines and library routines) and the amount of LOCAL-STORAGE data items that are used. This can have a significant impact on the CPU time of an application that is call intensive. You should not use STORAGE(,,00) to initialize variables for your application. Instead, you should change your application to initialize their own variables. You should not use STORAGE(,,00) in any performance-critical application.

Performance considerations using STORAGE:

- On the average, STORAGE(00,00,00) was 11% slower than STORAGE(NONE,NONE,NONE), with a range of equivalent to 133% slower. One RENT program calling a RENT subprogram with a 40 MB WORKING-STORAGE was 28% slower. Note that when using call intensive applications, the degradation can be 200% slower or more.
- On the average, STORAGE(00,NONE,NONE) was equivalent to STORAGE(NONE,NONE,NONE). One RENT program calling a RENT subprogram with a 40 MB WORKING-STORAGE was 5% slower.
- On the average, STORAGE(NONE,00,NONE) was equivalent to STORAGE(NONE,NONE,NONE). One RENT program calling a RENT subprogram with a 40 MB WORKING-STORAGE was 9% slower.
- For a call intensive program, STORAGE(NONE,NONE,00) can degrade more than 100%, depending on the number of calls.

  Note: The call intensive tests measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

related references
STORAGE (*z/OS Language Environment Programming Reference*)
STORAGE (*z/OS Language Environment Customization*)
COBOL and Language Environment runtime options comparison (*z/OS Language Environment Runtime Application Migration Guide*)

TEST

Default
NOTEST(ALL,*,PROMPT,INSPPREF)

Recommended
NOTEST(ALL,*,PROMPT,INSPPREF)

Considerations

The TEST option specifies the conditions under which Debug Tool assumes control when the user application is invoked.

Since this may result in Debug Tool being initialized and invoked, there may be some additional overhead when using TEST. Specifying NOTEST will eliminate this overhead.

related references

TEST | NOTEST (*z/OS Language Environment Programming Reference*)
TEST | NOTEST (*z/OS Language Environment Customization*)

TRAP

Default

ON,SPIE

Recommended

ON,SPIE

Considerations

The TRAP option allows LE to intercept an abnormal termination (abend), provide the abend information, and then terminate the LE runtime environment.

TRAP(ON) also assures that all files are closed when an abend is encountered and is required for proper handling of the ON SIZE ERROR clause of arithmetic statements for overflow conditions. In addition, LE uses condition handling internally and requires TRAP(ON). TRAP(OFF) prevents LE from intercepting the abend. In general, there will not be any significant impact on the performance of a COBOL application when using TRAP(ON).

When using the SPIE suboption, LE will issue an ESPIE to handle program interrupts. When using the NOSPIE suboption, LE will handle program interrupts via an ESTAE.

Performance considerations using TRAP: On the average, TRAP(ON) was equivalent to TRAP(OFF).

related tasks

Closing QSAM files (*Enterprise COBOL for z/OS Programming Guide*)
Closing VSAM files (*Enterprise COBOL for z/OS Programming Guide*)
Closing line-sequential files (*Enterprise COBOL for z/OS Programming Guide*)
Handling errors in arithmetic operations
(*Enterprise COBOL for z/OS Programming Guide*)

related references

Language Environment runtime options
(*Enterprise COBOL for z/OS Migration Guide*)
TRAP effects on the condition handling process
(*z/OS Language Environment Programming Guide*)
TRAP runtime option and user-written condition handlers
(*z/OS Language Environment Programming Guide*)
TRAP runtime option and CEEBXITA
(*z/OS Language Environment Programming Guide*)
TRAP (*z/OS Language Environment Programming Reference*)
TRAP (*z/OS Language Environment Customization*)

VCTRSAVE

Default

OFF (Non-CICS); N/A (CICS)

Recommended

OFF (Non-CICS); N/A (CICS)

Considerations

The VCTRSAVE option specifies whether any language in the application uses the vector facility when the user-provided condition handlers are called.

If you do not have user-written condition handlers registered with CEEHDLR, or your user-written condition handlers do not get vector facility instructions generated by INSPECT statements with ARCH(11), then you should run with VCTRSAVE(OFF) to avoid this overhead.

Performance considerations using VCTRSAVE: On the average, VCTRSAVE(ON) was equivalent to VCTRSAVE(OFF).

related references

VCTRSAVE (*z/OS Language Environment Programming Reference*)
VCTRSAVE (*z/OS Language Environment Customization*)

Chapter 5. COBOL and LE features that affect runtime performance

COBOL and Language Environment have several installation and environment tuning features that can enhance the performance of your application.

The following information describes some additional factors that should be considered for the application.

Storage management tuning

Storage management tuning can reduce the overhead involved in getting and freeing storage for the application program. With proper tuning, several GETMAIN and FREEMAIN calls can be eliminated.

First of all, storage management was designed to keep a block of storage only as long as necessary. This means that during the execution of a COBOL program, if any block of storage becomes unused, it will be freed. This can be beneficial in a transaction environment (or any environment) where you want storage to be freed as soon as possible so that other transactions (or applications) can make efficient use of the storage.

However, it can also be detrimental if the last block of storage does not contain enough free space to satisfy a storage request by a library routine. For example, suppose that a library routine needs 2K of storage but there is only 1K of storage available in the last block of storage. The library routine will call storage management to request 2K of storage. Storage management will determine that there is not enough storage in the last block and issue a GETMAIN to acquire this storage (this GETMAINed size can also be tuned). The library routine will use it and then, when it is done, call storage management to indicate that it no longer needs this 2K of storage. Storage management, seeing that this block of storage is now unused, will issue a FREEMAIN to release the storage back to the operating system.

Now, if this library routine or any other library routine that needs more than 1K of storage is called often, a significant amount of CPU time degradation can result because of the amount of GETMAIN and FREEMAIN activity.

Fortunately, there is a way to compensate for this with LE; it is called storage management tuning. The RPTSTG(ON) runtime option can help you in determining the values to use for any specific application program. You use the value returned by the RPTSTG(ON) option as the size of the initial block of storage for the HEAP, ANYHEAP, BELOWHEAP, STACK, and LIBSTACK runtime options. This will prevent the above from happening in an all VS COBOL II, COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, or Enterprise COBOL application. However, if the application also contains OS/VS COBOL programs that are being called frequently, the RPTSTG(ON) option may not indicate a need for additional storage. Increasing these initial values can also eliminate some storage management activity in this mixed environment.

The IBM supplied default storage options for batch applications are listed below:

```
ANYHEAP(16K,8K,ANYWHERE,FREE)
BELOWHEAP(8K,4K,FREE)
HEAP(32K,32K,ANYWHERE,KEEP,8K,4K)
LIBSTACK(4K,4K,FREE)
STACK(128K,128K,ANYWHERE,KEEP,512K,128K)
THREADHEAP(4K,4K,ANYWHERE,KEEP)
THREADSTACK(OFF,4K,4K,ANYWHERE,KEEP,128K,128K)
```

If you are running only COBOL applications, you can do some further storage tuning as indicated below:

```
STACK(64K,64K,ANYWHERE,KEEP)
```

The IBM supplied default storage options for CICS applications are listed below:

```
ANYHEAP(4K,4080,ANYWHERE,FREE)
BELOWHEAP(4K,4080,FREE)
HEAP(4K,4080,ANYWHERE,KEEP,4K,4080)
```

```
LIBSTACK(32,4000,FREE)
STACK(4K,4080,ANYWHERE,KEEP,4K,4080)
```

If all of your applications are AMODE(31), you can use ALL31(ON) and STACK(,,ANYWHERE). Otherwise, you must use ALL31(OFF) and STACK(,,BELOW).

Overall below the line storage requirements have been reduced by reducing the default storage options and by moving some of the library routines above the line.

Note: Beginning with LE for z/OS Release 1.2, the runtime defaults have changed to ALL31(ON),STACK(,,ANY). LE for OS/390 Release 2.10 and earlier runtime defaults were ALL31(OFF),STACK(,,BELOW).

Storage tuning user exit

In an environment where Language Environment is being initialized and terminated constantly, such as CICS, IMS, or other transaction processing type of environments, tuning the storage options can improve the overall performance of the application.

This helps to reduce the GETMAIN and FREEMAIN activity. The Language Environment storage tuning user exit is one way that you can manage the task of selecting the best values for your environment. The storage tuning user exit allows you to set storage values for your main programs without having to linkedit the values into your load modules.

related references
Storage tuning user exit (*z/OS Language Environment Customization*)

Using the CEEENTRY and CEETERM macros

To improve the performance of non-LE-conforming Assembler calling COBOL, you can make the Assembler program LE-conforming. This can be done using the CEEENTRY and CEETERM macros provided with LE.

This helps to reduce the GETMAIN and FREEMAIN activity. The Language Environment storage tuning user exit is one way that you can manage the task of selecting the best values for your environment. The storage tuning user exit allows you to set storage values for your main programs without having to linkedit the values into your load modules.

Performance considerations using the CEEENTRY and CEETERM macros (measuring CALL overhead only):

- One testcase (an LE-conforming Assembler calling COBOL) using the CEEENTRY and CEETERM macros was 99% faster than not using them.

  Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

See also "First program not LE-conforming" on page 46 for additional performance considerations comparing using CEEENTRY and CEETERM with other environment initialization techniques.

related references
CEEENTRY macro (*z/OS Language Environment Programming Guide*)
CEETERM macro (*z/OS Language Environment Programming Guide*)

Using preinitialization services (CEEPIPI)

LE preinitialization services (CEEPIPI) can also be used to improve the performance of non-LE-conforming Assembler calling COBOL.

LE preinitialization services let an application initialize the LE environment once, execute multiple LE-conforming programs, then explicitly terminate the LE environment. This substantially reduces the use of system resources that would have been required to initialize and terminate the LE environment for each program of the application.

See “Using CEEPIPI with Call_Sub” for an example of using CEEPIPI to call a COBOL subprogram and “Using CEEPIPI with Call_Main” for an example of using CEEPIPI to call a COBOL main program.

Performance considerations using CEEPIPI (measuring CALL overhead only):

- One testcase (a non-LE-conforming Assembler calling COBOL) using CEEPIPI to invoke the COBOL program as a subprogram was 99% faster than not using CEEPIPI.
- The same program using CEEPIPI to invoke the COBOL program as a main program was 95% faster than not using CEEPIPI.

Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

See “First program not LE-conforming” on page 46 for additional performance considerations comparing using CEEPIPI with other environment initialization techniques.

related references
Using preinitialization services (*z/OS Language Environment Programming Guide*)

Using library routine retention (LRR)

LRR is a function that provides a performance improvement for those applications or subsystems running on MVS with the following attributes:

- The application or subsystem invokes programs that require LE
- The application or subsystem is not LE-conforming (i.e., LE is not already initialized when the application or subsystem invokes programs that require LE)
- The application or subsystem repeatedly invokes programs that require LE running under the same MVS task
- The application or subsystem is not using LE preinitialization services

LRR is useful for non-LE-conforming assembler drivers that repeatedly call LE-conforming languages and for IMS/TM regions. LRR is not supported under CICS. See “IMS” on page 48 for information on using LRR under IMS.

When LRR has been initialized, LE keeps a subset of its resources in memory after the environment terminates. As a result, subsequent invocations of programs in the same MVS task that caused LE to be initialized are faster because the resources can be reused without having to be reacquired and reinitialized. The resources that LE keeps in memory upon LE termination are:

- LE runtime load modules
- Storage associated with these load modules
- Storage for LE startup control blocks

When LRR is terminated, these resources are released from memory.

LE preinitialization services and LRR can be used simultaneously. However, there is no additional benefit by using LRR when LE preinitialization services are being used. Essentially, when LRR is active and a non-LE-conforming application uses preinitialization services, LE remains preinitialized between repeated invocations of LE-conforming programs and does not terminate. Upon return to the non-LE-conforming application, preinitialization services can be called to terminate the LE environment, in which case LRR will be back in effect. See “Using library routine retention (LRR)” on page 41 for an example of using LRR.

Performance considerations using LRR:

- One testcase (a non-LE-conforming Assembler calling COBOL) using LRR was 96% faster than not using LRR.

  Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

See “First program not LE-conforming” on page 46 for additional performance considerations comparing using LRR with other environment initialization techniques.

related references
Language Environment library routine retention (LRR)
(*z/OS Language Environment Programming Guide*)
Using Language Environment under IMS
(*z/OS Language Environment Customization*)

Library in the LPA/ELPA

Placing the COBOL and the LE library routines in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) can also help to improve total system performance.

This will reduce the real storage requirements for the entire system for COBOL/370, COBOL for MVS & VM, COBOL for OS/390 & VM, Enterprise COBOL, VS COBOL II RES, or OS/VS COBOL RES applications since the library routines can be shared by all applications instead of each application having its own copy of the library routines. For a list of COBOL library routines that are eligible to be placed in the LPA/ELPA, see members CEEWLPA and IGZWMLP4 in the SCEESAMP data set.

Placing the library routines in a shared area will also reduce the I/O activity since they are loaded only once when the system is started and not for each application program.

related references
Modules eligible for the link pack area
(*z/OS Language Environment Customization*)
Planning to link and run with Language Environment
(*z/OS Language Environment Runtime Application Migration Guide*)

Using CALLs

You should consider storage management tuning for all CALL intensive applications.

With static CALLs (call literal with NODYNAM), all programs are link-edited together, and hence, are always in storage, even if you do not call them. However, there is only one copy of the bootstrapping library routines link-edited with the application. With dynamic CALLs (call literal with DYNAM or call identifier), each subprogram is link-edited separately from the others. They are brought into storage only if they are needed. This is the better way of managing complicated applications. However, each subprogram has its own copy of the bootstrapping library routines link-edited with it, bringing multiple copies of these routines in storage as the application is executing.

Another aspect is program loading. Since a dynamic CALL subprogram is brought into storage when it is first needed, it is not loaded into storage at the beginning together with the caller program. There is an overhead in terms of program load processing. In general, it is beneficial to use dynamic calls when the call structure of an application is complicated, the size of the subprograms is not small, and not all subprograms are called in a particular run of the application.

Performance considerations for using CALLs between programs compiled with similar COBOL versions (measuring CALL overhead only):

- Static CALL literal was on average 40% faster than dynamic CALL literal.
- Static CALL literal was on average 52% faster than dynamic CALL identifier.
- Dynamic CALL literal was on average 20% faster than dynamic CALL identifier.

Note:

- For the purpose of this discussion, the following COBOL versions are considered similar:
  - COBOL V4.2 and prior releases
  - COBOL V5.1 and later releases
- These measurements are only for the overhead of the CALL (i.e. the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms may have different results.

related tasks
Transferring control to another program
(*Enterprise COBOL for z/OS Programming Guide*)

Using IS INITIAL on the PROGRAM-ID statement or INITIAL compiler option

The IS INITIAL clause on the PROGRAM-ID statement or the INITIAL compiler option specifies that when a program is called, it and any programs that it contains will be entered in their initial or first-time called state.

There is an overhead in initializing all WORKING-STORAGE variables with VALUE clauses. The performance impact depends on the number and sizes of such variables.

Using IS RECURSIVE on the PROGRAM-ID statement

The IS RECURSIVE clause on the PROGRAM-ID statement specifies that the COBOL program can be recursively called while a previous invocation is still active.

The IS RECURSIVE clause is required for all programs that are compiled with the THREAD compiler option.

Performance considerations for using IS RECURSIVE on the PROGRAM-ID statement (measuring CALL overhead only):

- One testcase (an LE-conforming Assembler repeatedly calling COBOL) using IS RECURSIVE was 15 % slower than not using IS RECURSIVE.

  Note: This test measured only the overhead of the CALL (i.e., the subprogram did only a GOBACK); thus, a full application that does more work in the subprograms is not degraded as much.

Chapter 6. Other product related factors that affect runtime performance

It is important to understand COBOL's interaction with other products in order to enhance the performance of your application.

This section describes some product related factors that should be considered for the application.

Decimal overflow implications in ILC applications

Interlanguage communication (ILC) applications are applications built of two or more high-level languages (such as COBOL, PL/I, or C) and frequently assembler.

ILC applications run outside of the realm of a single language environment, which creates special conditions, such as how the data from each language maps across load module boundaries, how conditions are handled, or how data can be passed and received by each language.

While LE fully supports ILC applications, there can be significant performance implications when using them with COBOL applications. One area to look at is condition handling.

COBOL normally either ignores decimal overflow conditions or handles them by checking the condition code after the decimal instruction. However, when languages such as C or PL/I are in the same application as COBOL, these decimal overflow conditions will now be handled by LE condition management since both C and PL/I set the Decimal Overflow bit in the program mask. This can have a significant impact on the performance of the COBOL application, especially under CICS, if decimal overflows are occurring during arithmetic operations in the COBOL program.

The following programming languages or COBOL features require hardware decimal overflows to be enabled. During arithmetic operations in COBOL compile units, Language Environment and COBOL runtime condition management work together to resume execution after overflows in COBOL compile units. This can have a significant impact on the performance of the COBOL application, especially under CICS, if many decimal overflows occur during arithmetic operations in COBOL compile units.

- Interlanguage Communication (ILC) with C/C++ and PL/I
- DLL calls
- XML GENERATE and XML PARSE
- JSON GENERATE
- BPXWDYN dynamic allocation

Notes:

- These languages or features need not be used to affect decimal overflows, their mere presence in the executable code will enable decimal overflows at the time the executable is loaded such as via a dynamic call.
- The C/C++ or PL/I program does not need to be called. Just the presence of C/C++ or PL/I in the load module will cause this degradation for decimal overflow conditions.
- The DLL implementation uses the C/C++ runtime.
- When XML GENERATE, XML PARSE, or JSON GENERATE statements are in the program, the C runtime library will be initialized. Hence, the decimal overflow bit in the program mask will be set even though you do not explicitly have a C or PL/I program in the application. This will also cause the same degradation for decimal overflow conditions. JSON PARSE statement does not require the C runtime library. Therefore, JSON PARSE does not affect the decimal overflow bit in the program mask.
- For the BPXWDYN interface to dynamic allocation, use the alternate entry point BPXWDY2 instead. BPXWDY2 will preserve the program mask.

For a COBOL program using COMP-3 (PACKED-DECIMAL) data types in 100,000 arithmetic statements that cause a decimal overflow condition, the C or PL/I, ILC case was over 99.98% slower than the COBOL-only, non-ILC case.

Related references
Performance of decimal overflows (*Enterprise COBOL for z/OS Migration Guide*)

First program not LE-conforming

If the first program in the application is non LE-conforming, and if this program is repeatedly calling COBOL, there can be a significant degradation because the COBOL environment must be initialized and terminated each time a COBOL main program is invoked.

This overhead can be reduced by doing one of the following (listed in order of most improvement to least improvement):

- Use the CEEENTRY and CEETERM macros in the first program of the application to make it an LE-conforming program.
- Call the first program of the application from a COBOL stub program (a program that just has a call statement to the original first program).
- Call CEEPIPI sub from the first program of the application to initialize the LE environment, invoke the COBOL program, and then terminate the LE environment when the application is complete.
- Use the runtime option RTEREUS to initialize the runtime environment for reusability, making all COBOL main programs become subprograms.
- Use the Library Routine Retention (LRR) function (similar to the function provided by the LIBKEEP runtime option in VS COBOL II).
- Call CEEPIPI main from the first program of the application to initialize the LE environment, invoke the COBOL program, and then terminate the LE environment when the application is complete.
- Place the LE library routines in the LPA or ELPA. The list of routines to put in the LPA or EPLA is release dependent and is the same routines listed under the IMS preload list considerations.

related references
Assembler considerations (*z/OS Language Environment Programming Guide*)

CICS

Language Environment uses more transaction storage than VS COBOL II. This is especially noticeable when more than one run-unit (enclave) is used since storage is managed at the run-unit level with LE. This means that HEAP, STACK, ANYHEAP, etc. are allocated for each run-unit under LE. With VS COBOL II, stack (SRA) and heap storage are managed at the transaction level. Additionally, there are some LE control blocks that need to be allocated.

In order to minimize the amount of below the line storage used by LE under CICS, you should run with ALL31(ON) and STACK(,,ANYWHERE) as much as possible. In order to do this, you have to identify all of your AMODE(24) COBOL programs that are not OS/VS COBOL. Then you can either make the necessary coding changes to make them AMODE(31) or you can link-edit a CEEUOPT with ALL31(OFF) and STACK(,,BELOW) as necessary for those run units that need it. You can find out how much storage a particular transaction is using by looking at the auxiliary trace data for that transaction. You do not need to be concerned about OS/VS COBOL programs since the LE runtime options do not affect OS/VS COBOL programs running under CICS. Also, if the transaction is defined with TASKDATALOC(ANY) and ALL31(ON) is being used and the programs are compiled with DATA(31), then LE does not use any below the line storage for the transaction under CICS, resulting in some additional below the line storage savings.

There are two CICS SIT options that can be used to reduce the amount of GETMAIN and FREEMAIN activity, which will help the response time. The first one is the RUWAPOOL SIT option. You can set RUWAPOOL to YES to reduce the GETMAIN and FREEMAIN activity. The second is the AUTODST SIT option. If you are using CICS Transaction Server Version 1 Release 3 or later, you can also set AUTODST

to YES to cause Language Environment to automatically tune the storage for the CICS region. Doing this should result in fewer GETMAIN and FREEMAIN requests in the CICS region. Additionally, when using AUTODST=YES, you can also use the storage tuning user exit (see "Storage tuning user exit" on page 40) to modify the default behavior of this automatic storage tuning.

For details, see Using Language Environment under CICS in the *z/OS Language Environment Customization*.

The RENT compiler option is required for an application running under CICS. Additionally, if the program is run through the CICS translator or co-processor (i.e., it has EXEC CICS commands in it), it must also use the NODYNAM compiler option. CICS Transaction Server 1.3 or later is required for Enterprise COBOL.

For details, see *RENT* in the *Enterprise COBOL for z/OS Programming Guide*.

Enterprise COBOL supports static and dynamic calls to Enterprise COBOL and VS COBOL II (with the RES option) subprograms containing CICS commands or dependencies. Note that Enterprise COBOL 4.2 and earlier releases also support calls to VS COBOL II with the NORES option. Static calls are done with the CALL literal statement and dynamic calls are done with the CALL identifier statement. Converting EXEC CICS LINKs to COBOL CALLs can improve transaction response time and reduce virtual storage usage. Enterprise COBOL does not support calls to or from OS/VS COBOL programs in a CICS environment. In this case, EXEC CICS LINK must be used.

Note: When using EXEC CICS LINK under Language Environment, a new run-unit (enclave) will be created for each EXEC CICS LINK. This means that new control blocks will be allocated and subsequently freed for each LINKed to program. This will result in an increase in the number of storage requests. If storage management tuning has not been done, you may experience more storage requests per enclave. As a result of a new enclave being created for each EXEC CICS LINK, the CPU time performance will also be degraded when compared to VS COBOL II. If your application uses many EXEC CICS LINKs, you can avoid this extra overhead by using COBOL CALLs whenever possible.

If you are using the COBOL CALL statement to call a program that has been translated with the CICS translator or has been compiled with the CICS co-processor, you must pass DFHEIBLK and DFHCOMMAREA as the first two parameters on the CALL statement. However, if you are calling a program that has not been translated, you should not pass DFHEIBLK and DFHCOMMAREA on the CALL statement. Additionally, if your called subprogram does not use any of the EXEC CICS condition handling commands, you can use the runtime option CBLPSHPOP(OFF) to eliminate the overhead of doing an EXEC CICS PUSH HANDLE and an EXEC CICS POP HANDLE that is done for each call by the LE runtime. The CBLPSHPOP setting can be changed dynamically by using the CLER transaction.

As long as your usage of all binary (COMP) data items in the application conforms to the PICTURE and USAGE specifications, you can use TRUNC(OPT) to improve transaction response time. This is recommended in performance sensitive CICS applications. If your usage of any binary data item does not conform to the PICTURE and USAGE specifications, you can either use a COMP-5 data type or increase the precision in the PICTURE clause instead of using the TRUNC(BIN) compiler option. Note that the CICS translator does not generate code that will cause truncation and the CICS co-processor uses COMP-5 data types which does not cause truncation. If you were using NOTRUNC with your OS/VS COBOL programs without problems, TRUNC(OPT) on IBM Enterprise COBOL behaves in a similar way. For additional information on the TRUNC compiler option, see *TRUNC* in the *Enterprise COBOL for z/OS Programming Guide*.

Db2

As long as your usage of all binary (COMP) data items in the application conforms to the PICTURE and USAGE specifications and your binary data was created by COBOL programs, you can use TRUNC(OPT) to improve performance under Db2®.

This is recommended in performance sensitive Db2 applications. If your usage of any binary data item does not conform to the PICTURE and USAGE specifications, you should use COMP-5 data types or use the TRUNC(BIN) compiler option. If you were using NOTRUNC with your OS/VS COBOL programs without problems, TRUNC(OPT) on COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL

behaves in a similar way. For additional information on the TRUNC option, please refer to "TRUNC" on page 27.

The RENT compiler option must be used for COBOL programs used as Db2 stored procedures.

For the best performance, make sure that you use codepages that are compatible to avoid unnecessary conversions. For example, if your Db2 database uses codepage 037, but you use the CODEPAGE(1140), SQL, SQLCCSID compiler options, the performance can be slower than using either CODEPAGE(037), SQL, SQLCCSID or CODEPAGE(1140), SQL, NOSQLCCSID since the first set of options require conversions to match the codepage but the second and third set of options do not require such conversions.

DFSORT

Use the FASTSRT compiler option to improve the performance of most sort operations. With FASTSRT, the DFSORT product performs the I/O on input and/or output files named in either or both of the SORT ... USING or SORT ... GIVING statements. If you have an INPUT PROCEDURE phrase or an OUTPUT PROCEDURE phrase for your sort files, the FASTSRT option has no impact to the INPUT PROCEDURE or the OUTPUT PROCEDURE. However, if you have an INPUT PROCEDURE phrase with a GIVING phrase or a USING phrase with an OUTPUT PROCEDURE phrase, FASTSRT will still apply to the USING or GIVING part of the SORT statement. The complete list of requirements is contained in the *Enterprise COBOL for z/OS Programming Guide*.

Performance considerations using DFSORT:

- One program that processed 100,000 records is 50% faster when using FASTSRT compared to using NOFASTSRT.

related references
FASTSRT (*Enterprise COBOL for z/OS Programming Guide*)

IMS

If the application is running under IMS, preloading the application program and the library routines can help to reduce the load/search overhead, as well as reduce the I/O activity.

This is especially true for the library routines since they are used by every COBOL program. When the application program is preloaded, subsequent requests for the program are handled faster because it does not have to be fetched from external storage. The RENT compiler option is required for preloaded applications.

Using the Library Routine Retention (LRR) function can significantly improve the performance of COBOL transactions running under IMS/TM. LRR provides function similar to that of the VS COBOL II LIBKEEP runtime option. It keeps the LE environment initialized and retains in memory any loaded LE library routines, storage associated with these library routines, and storage for LE startup control blocks. To use LRR in an IMS dependent region, you must do the following steps:

1. In your startup JCL or procedure to bring up the IMS dependent region, specify the PREINIT=xx parameter, where xx is the 2-character suffix of the DFSINTxx member in your IMS PROCLIB data set.
2. Include the name CEELRRIN in the DFSINTxx member of your IMS PROCLIB data set.
3. Bring up your IMS dependent region.

You can also create your own load module to initialize the LRR function by modifying the CEELRRIN sample source in the SCEESAMP data set. If you do this, use your module name in place of CEELRRIN above.

Warning: The RTEREUS option is not recommended for IMS and its usage should be avoided. If the RTEREUS runtime option is used, the top level COBOL programs of all applications must be preloaded.

Using RTEREUS will keep the LE environment up until the region goes down or until a STOP RUN is issued by a COBOL program. This means that every program and its WORKING-STORAGE (from the time the first COBOL program was initialized) is kept in the region. Although this is very fast, you may find that

the region may soon fill to overflowing, especially if there are many different COBOL programs that are invoked.

Note: Refer to *z/OS Language Environment Customization* for restrictions and usage notes about this option.

When not using RTEREUS or LRR, it is recommended that you preload the following library modules:

- For all COBOL applications: CEEBINIT, IGZCPAC, IGZCPCO, CEEEV005, CEEPLPKA, IGZETRM, IGZEINI, IGZCLNK, CEEEV004, IGZXDMR, IGZXD24, IGZXLPIO, IGZXLPKA, IGZXLPKB, IGZXLPKC
- If the application also contains VS COBOL II programs: IGZCTCO, IGZEPLF, and IGZEPCL

Preloading should reduce the amount of I/O activity associated with loading and deleting these modules for each transaction.

Other than the COBOL library modules listed above, you should also preload any of the below the line routines that you need. A list of the below the line routines can be found in the Language Environment Customization manual.

Additionally, heavily used application programs can be compiled with the RENT compiler option and preloaded to reduce the amount of I/O activity associated with loading them.

The TRUNC(OPT) compiler option can be used if the following conditions are satisfied:

- You are not using a database that was built by a non-COBOL program.
- Your usage of all binary data items conforms to the PICTURE and USAGE specifications for the data items (e.g., no pointer arithmetic using binary data types).

Otherwise, you should use the TRUNC(BIN) compiler option or COMP-5 data types. For additional information on the TRUNC compiler option, see "TRUNC" on page 27.

related references
Language Environment library routine retention (LRR)
(*z/OS Language Environment Programming Guide*)
Using Language Environment under IMS
(*z/OS Language Environment Customization*)
Language Environment COBOL component modules
(*z/OS Language Environment Customization*)

LLA

Enterprise COBOL programs (V5 and later releases) linking with CSECTs that have the RMODE 24 attribute may be excluded from management by Library Lookaside (LLA).

Considerations when using the LLA facility

Programs in the following list contain CSECTs with the RMODE 24 attribute:

- Enterprise COBOL program that is compiled with the RMODE(24) or NORENT compiler options.
- VS COBOL II program that is compiled with the NORENT compiler option.
- Assembler program that contains CSECT with RMODE 24.

By default, the RMODE attribute of an Enterprise COBOL V5 (or later) program is RMODE ANY. When such program is linked with any of the above, the binder will place RMODE 24 CSECTS in one segment, and the Enterprise COBOL V5 code in a second segment. There is also a third segment for the C-WSA class (new starting in COBOL V5). Program objects with more than two segments cannot be used with the Library Lookaside (LLA) facility. This issue can be avoided by specifying the RMODE(24) compiler option explicitly for the Enterprise COBOL V5 program, and specifying the DYNAM=NO binder option (default for the binder). This would make the RMODE attributes consistent within the program object. Alternatively, you can also change the compilation of the COBOL programs in the above list by not using the RMODE(24) and NORENT options, and not having RMODE 24 CSECTs in assemble programs.

Chapter 7. Coding techniques to get the most out of V6

This section focuses on how the source code can be modified to tune a program for better performance. Coding style, as well as data types, can have a significant impact on the performance of an application.

BINARY (COMP or COMP-4)

BINARY data and synonyms COMP and COMP-4 are the two's complement data representation in COBOL.

BINARY data is declared with a PICTURE clause as with internal or external decimal data but the underlying data representation is as a halfword (2 bytes), a fullword (4 bytes) or a doubleword (8 bytes).

The compiler option TRUNC(OPT | STD | BIN) determines if and how the compiler corrects values back to the declared picture clause and how much significant data is present when accessing a data item.

Although the overall general performance considerations for BINARY data and the TRUNC option from V4 and earlier versions still apply in V6, some of the relative performance differences for the various TRUNC suboptions have changed (sometimes dramatically). These changes may impact coding and compiler option choices.

To quantify the relative and absolute performance differences a series of addition operations on binary data items were executed in a loop on a z13 machine. The 4 tests below contain the same type and number of arithmetic operations but have a varying number of digits. All of the operands are signed.

- TEST 1: 8 additions with one each of 1 through 8 digits
- TEST 2: 8 additions each with 9 digits
- TEST 3: 8 additions with one each of 10 through 17 digits
- TEST 4: 8 additions each with 18 digits

The tests were then compiled varying the TRUNC option.

The first experiment specified the TRUNC(STD) compiler option.

TRUNC(STD) instructs the compiler to always correct back to the specified PICTURE clause and allows the compiler to assume that loaded values only have the specified number of PICTURE clause digits.

Table 5. Performance differences results of four test cases when specifying TRUNC(STD)

TRUNC(STD)	**V4 versus TEST 1**	**V6 versus TEST 1**	**V6 versus V4**
TEST 1: 1-8 digits	100%	100%	16.6%
TEST 2: 9 digits	145.9%	116.9%	13.3%
TEST 3: 10-17 digits	479.1%	116.4%	4%
TEST 4: 18 digits	768.2%	100.5%	2.2%

These results demonstrate that:

- V6 outperforms V4 for all lengths when using TRUNC(STD).
- Although performance slows as the number of digits increases for both V4 and V6 it slows much more gradually and to much lower overall amount using V6 compared to V4.

The second experiment specified the TRUNC(BIN) compiler option. Specifying this option is equivalent to using the COMP-5 type for all BINARY data.

TRUNC(BIN) instructs the compiler to allow values to only correct back to the underlying data representation (two, four or eight bytes) instead of back to the specified PICTURE clause. This option

also requires the compiler to assume that loaded values can have up to two, four or eight bytes worth of significant data.

Table 6. Performance differences results of four test cases when specifying TRUNC(BIN)

TRUNC(BIN)	V4 versus TEST 1	V6 versus TEST 1	V6 versus V4
TEST 1: 1-8 digits	100%	100%	62.5%
TEST 2: 9 digits	154.5%	100.4%	40.6%
TEST 3: 10-17 digits	5098.2%	1925.5%	23.6%
TEST 4: 18 digits	5099.2%	1928%	23.6%

These results demonstrate that:

- V6 also outperforms V4 for all lengths when using TRUNC(BIN).
- V6 shows no slow down when testing at 9 digits.
- There is a dramatic reduction in performance for both V4 and V6 (but more so for V4 in absolute terms) when the length is increased beyond 9 digits. This is due to the TRUNC(BIN) requirement that input data may contain up to the full integer half/full/double word of data (and extra data type conversions and library routines are required).

The third and final experiment specified the TRUNC(OPT) compiler option. TRUNC(OPT) is a performance option. The compiler assumes that input data conforms to the PICTURE clause and then allows the compiler the freedom to manipulate data in either of the following ways that are most optimal:

- Correcting back to the PICTURE clause as with TRUNC(STD) or
- Only correcting back to the two, four or eight byte boundary as with TRUNC(BIN)

Table 7. Performance differences results of four test cases when specifying TRUNC(OPT)

TRUNC(OPT)	V4 versus TEST 1	V6 versus TEST 1	V6 versus V4
TEST 1: 1-8 digits	100%	100%	97%
TEST 2: 9 digits	333.3%	69.7%	20.4%
TEST 3: 10-17 digits	209.7%	70.4%	32.7%
TEST 4: 18 digits	5553.4%	129.7%	2.3%

These results demonstrate that:

- V6 also outperforms V4 when using TRUNC(OPT).
- V6 shows no slow down until the 18 digit case but still vastly outperforms V4 at this longest length.

The performance results are compared when using TRUNC(OPT), TRUNC(STD), and TRUNC(BIN) for V6 :

Table 8. Performance differences results when specifying TRUNC(OPT), TRUNC(STD), and TRUNC(BIN) for V6

V6	TRUNC(STD) vs TRUNC(OPT)	TRUNC(BIN) vs TRUNC(OPT)
TEST 1: 1-8 digits	109.1%	94.1%
TEST 2: 9 digits	172.4%	99.7%
TEST 3: 10-17 digits	178.5%	3864.7%
TEST 4: 18 digits	105.5%	2116.1%

Note: Use the TRUNC(OPT) only if you are sure the data being moved in the binary areas conforms to the PICTURE clause otherwise unpredictable results could occur. See *TRUNC* in the *Enterprise COBOL for z/OS Programming Guide* for more information.

Across all the TRUNC options and data item lengths just presented V6 outperforms V4. These improvements are due to the following reasons:

- The use of 64-bit 'G' form instructions enables much more efficient code for > 8 digit cases
- More efficient library routines for the very large TRUNC(BIN) cases

Chapter 2, "Prioritizing your application for migration to V6," on page 7 has a specific example of binary double word arithmetic (Large Binary Arithmetic) that demonstrates the performance improvement for this type of operation relative to version 4 of the compiler. In this example V6 is considerably faster than V4 and earlier compiler releases.

The relative performance differences across the different TRUNC options have been smoothed out compared to V4. This is primarily due to the compiler inserting a runtime test for overflow. If no overflow is possible then an expensive 'divide' hardware instruction is avoided.

If your data is known to conform to the PICTURE clause then TRUNC(OPT) remains the best overall option to choose but relatively speaking it improves less over TRUNC(STD) than in V4 and overall absolute performance is better with either option in V6.

Although TRUNC(BIN) enables more efficient code when storing out a COMPUTE or MOVE result it continues to significantly harm the performance when these data items are used as input to arithmetic statements (as the compiler must assume the max 2,4,8 byte size). V6 optimizes the correction code for TRUNC(STD) so the performance benefit of TRUNC(BIN) has been reduced slightly.

It might be better to only specify COMP-5 for select data items versus using TRUNC(BIN). For example, performance will usually be improved if data items in COMPUTE statements in particular are not specified with COMP-5.

DISPLAY

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "Avoid using USAGE DISPLAY data items for computations (especially in areas that are heavily used for computations)". This continues to be the best practice in V6. However, using the options OPT(1 | 2) and ARCH(10 | 11) enables the V6 compiler to efficiently convert DISPLAY operands to Decimal Floating Point (DFP). This optimization reduces the overhead of using DISPLAY data items in computations.

Comparing data USAGE DISPLAY:

```
1 A pic s9(17).
1 B pic s9(17).
1 C pic s9(18).
```

To COMP-3:

```
1 A pic s9(17) COMP-3.
1 B pic s9(17) COMP-3.
1 C pic s9(18) COMP-3.
```

For the statement:

```
ADD A TO B GIVING C.
```

In V4 using COMP-3 is 22% faster than using DISPLAY; while in V6 using COMP-3 is only 4% faster than using DISPLAY. So the DISPLAY performance in this case has improved by 18% in V6 compared to V4.

Although performance comparisons will vary for different sizes of data and different types of computational statements the performance of DISPLAY data items in computations has generally improved in V6 when using ARCH(10) and OPT(1 | 2). Despite the improvements between V6 and V4, it is still recommended to use COMP-3 or BINARY for data items that will be used in computations. In

particular, if a data item will be used as a loop counter or a table index, converting that data item from DISPLAY to BINARY can lead to significant performance gains.

PACKED-DECIMAL (COMP-3)

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "When using PACKED-DECIMAL (COMP-3) data items in computations, use 15 or fewer digits in the PICTURE specification to avoid the use of library routines for multiplication and division".

Using V6 and the options ARCH(8 | 9 | 10 | 11) and OPT(1 | 2), the compiler can generate inline decimal floating-point (DFP) code for some of these larger multiplication and division operations. The maximum intermediate result size supported for this optimization is 34 digits. Although there is some overhead in this conversion to DFP, it is less of a penalty than having to invoke a library routine. This is also true for external decimal (DISPLAY and NATIONAL) types that are converted by the compiler to packed decimal for COMPUTE statements.

Using V6.2 and ARCH(12), the compiler instead uses the vector packed-decimal facility, which accelerates packed and zoned decimal computation by storing intermediate results in vector registers instead of in memory. This avoids the overhead of conversion to DFP.

Fixed-point versus floating-point

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "When using fixed-point exponentiations with large exponents, the calculation can be done more efficiently by using operands that force the exponentiation to be evaluated in floating-point".

In V6, it is still true that floating point exponentiation is much faster than fixed-point exponentiation; however, the relative cost of each type of exponentiation has changed from V4 to V6.

Consider the following code example:

```
01 A PIC S9(6)V9(12) COMP-3 VALUE 0.
01 B PIC S9V9(12) COMP-3 VALUE 1.234567891.
01 C PIC S9(10) COMP-3 VALUE -9.

COMPUTE A = (1 + B) ** C. (original)
COMPUTE A = (1.0E0 + B) ** C. (forced to floating-point)
```

The original, fixed-point exponentiation, is 93% faster in V6 compared to V4.

The forced to floating point exponentiation is 48% faster in V6 compared to V4.

However, because floating point exponentiation remains many times faster than fixed-point exponentiation, it is still recommended to use floating point exponentiation whenever possible.

Factoring expressions

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "For evaluating arithmetic expressions, the compiler is bound by the left-to-right evaluation rules for COBOL. In order for the optimizer to recognize constant computations (that can be done at compile time) or duplicate computations (common subexpressions), move all constants and duplicate expressions to the left end of the expression or group them in parentheses."

The V6 compiler factors expressions as a part of optimization and no longer requires this type of factoring to be done at the source code level as was recommended in V4. However, this only applies within a single expression. Consider the following example, which shows two ways of summing the total cost of a set of items and applying a discount:

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM
```

```
MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT
```

The first block of code performs ten multiplications, while the second block of code only performs one. The second block of code is therefore more efficient. The V6 compiler does not perform this type of factoring across loops. It's done at the source code level.

Symbolic constants

In *IBM Enterprise COBOL Version 4 Release 2 Performance Tuning*, it says: "If you want the optimizer to recognize a data item as a constant throughout the program, initialize it with a VALUE clause and don't modify it anywhere in the program".

This remains a valid and important recommendation, with one difference. The V6 compiler tolerates the data item being reinitialized to an identical value as was specified in the VALUE clause. In this case, the compiler recognizes that the value of the data item has not changed from its initial value and will still treat it as a constant.

Performance tuning considerations for Occurs Depending On tables

Usually the relative ordering of data item declarations does not have a significant or easily predictable impact on performance.

However, if your program contains Occurs Depending On (ODO) tables, the specific group layout of data items that follow an ODO table might lead to greatly degraded performance when accessing certain other variables.

Consider an ODO table declared as below:

```
01 TABLE-1.
   05 X PIC S9(4) comp.
   05 Y OCCURS 3 TIMES
        DEPENDING ON X PIC X.
   05 Z PIC S9.
```

Because the size of item Y in TABLE-1 depends on another data-item, any subsequent non-subordinate items in the same level-01 record are variably located items, such as item Z in the previous example.

Any load or store to the variably located item Z requires additional code to be generated by the compiler to determine the location of Z based on the current value of X. If ODO tables are nested then multiple extra computations are required.

However, by always ending a record after the ODO table, all variables declared after the table will no longer be variably located and access to these variables will be much more efficient. The following example adds a new level 01 record after the ODO table and before any other variables are declared:

```
01 TABLE-1.
   05 X PIC S9(4) comp.
   05 Y OCCURS 3 TIMES
        DEPENDING ON X PIC X.
01 WS-VARS.
   05 Z PIC S9.
```

Using PERFORM

Enterprise COBOL allows you to use the PERFORM verb in two basic ways: you may write an inline PERFORM or an out-of-line PERFORM.

An inline PERFORM is preferable from a performance perspective, because at all optimization levels control flow is straightforward. In addition, with V6 program objects, Debug Tool is capable of skipping over the contents of an out-of-line PERFORM. However, it is generally not desirable to replicate large or complicated code sequences simply to have inline PERFORMs.

In general, the executed code for an out-of-line PERFORM includes the following steps:

1. Establishing the program address where control will return when the PERFORM is completed and saving that address in a compiler generated LOCAL-STORAGE data item.
2. Branching to the start of the PERFORMed range.
3. Executing the PERFORMed range.
4. Branching back indirectly via the compiler generated data item mentioned in the first step.

In addition, the logic associated with phrases such as those for specifying the number of iterations or testing conditions is also executed.

At optimization levels above OPT(0), the compiler will attempt to remove some of the out-of-line branching code. If necessary, it will replicate code sequences to achieve this. This replication is limited to a maximum size for a PERFORM range and to a total maximum size for the whole program. There are no configuration options to control these maximum values.

This 'PERFORM inlining' optimization can be done on a per PERFORM statement basis. However, the nature of the range being PERFORMed must have certain characteristics in order to be a candidate.

In essence, out-of-line PERFORM statements should resemble procedure calls to have the best chance to be optimized. And, of course, that implies that the range being performed should resemble a procedure.

Typically, a procedure has a single entry point and control always returns ultimately to the caller. Therefore, in a performed range, all branching (except for additional PERFORM statements that code in the range itself might execute) should remain within the range. Similarly, the program should not contain branches (other than the PERFORMs of the range) from outside the performed range to statements inside it. For example, assuming that we have sequential sentences A, B and C in order in the program, the compiler does not optimize the following PERFORMs as the second PERFORM essentially branches into the middle of the first PERFORM's range:

```
PERFORM A THROUGH C
PERFORM B THROUGH C
```

Overlapping performed ranges in general can also inhibit the PERFORM inlining optimization as well as other global optimizations that tend to work best on more straightforward control flow constructs. Assuming that, in addition to sentences A, B and C, we also have (immediately following C) sentence D. The following statements result in overlapping performed ranges:

```
PERFORM A THROUGH C
PERFORM B THROUGH D
```

Recursively performed ranges are also not recommended in COBOL as these will also inhibit optimizations. For example, the following is not recommended:

```
A.            IF COND THEN PERFORM A.
```

Recursion can be more subtle than this case. Ranges A and B might recursively call each other and this would inhibit optimization.

In general, any branching between code in the main program and code in declarative sections (except the branching that happens as part of the natural flow of the COBOL program) is an impediment to optimization. And this is certainly true of branching in the form of PERFORM statements.

You should write the COBOL code that most naturally expresses the required logic. Sometimes, however, you can achieve the same thing in a number of ways, especially in utility routines. For example, the following code expresses logic one way:

```
LOCAL-STORAGE SECTION.
    01 ACTION PIC 9.

    PROCEDURE DIVISION.

    MOVE 1 TO ACTION
    PERFORM A

    MOVE 2 TO ACTION
    PERFORM A

    MOVE 1 TO ACTION
    PERFORM A


A.      IF ACTION = 1 DISPLAY "X" ELSE DISPLAY "Y".
```

In a case like this, it is likely beneficial to specialize the performed range as follows:

```
PERORM A1

PERFORM A2

PERFORM A1

A1. DISPLAY "X".
A2. DISPLAY "Y".
```

In this specific case, the optimizer will be able to achieve the same effect. It will start by replicating the statement in A at each PERFORM statement. Then it will have to spend compilation resources at each PERFORM statement to realize that, in each context, it can clearly identify whether or not ACTION = 1. Furthermore, in similar code patterns, there may be cases where the programmer knows something about the use of a utility range that the optimizer is not able to deduce.

Using QSAM files

When using QSAM files, use large block sizes whenever possible by using the BLOCK CONTAINS clause on your file definitions (the default with COBOL is to use unblocked files).

You can have the system determine the optimal block size for you by specifying the BLOCK CONTAINS 0 clause for any new files that you are creating and omitting the BLKSIZE parameter in your JCL for these files. You can also omit the BLOCK CONTAINS clause for the file and use the BLOCK0 compiler option to achieve the same effect. This should significantly improve the file processing time (both in CPU time and elapsed time).

Performance considerations using I/O buffers for a program that reads 14,000 records and wrote 28,000 records with no BLOCK CONTAINS clause and no BLKSIZE in the JCL:

- Using BLOCK0 was 90% faster and used 98% fewer EXCPs than NOBLOCK0.

Additionally, increasing the number of I/O buffers for heavy I/O jobs can improve both the CPU and elapsed time performance, at the expense of using more storage. This can be accomplished by using the BUFNO subparameter of the DCB parameter in the JCL or by using the RESERVE clause of the SELECT statement in the FILE-CONTROL paragraph. Note that if you do not use either the BUFNO subparameter or the RESERVE clause, the system default will be used.

Performance considerations using I/O buffers for a program that reads 14,000 records and wrote 28,000 records with no blocking:

- Using DCB=BUFNO=1 took 0.452 CPU seconds
- Using DCB=BUFNO=5 took 0.129 CPU seconds
- Using DCB=BUFNO=10 took 0.089 CPU seconds
- Using DCB=BUFNO=25 took 0.067 CPU seconds

Refer to Chapter 4.1 for a discussion on the location of QSAM buffers.

Using variable-length files

When writing to variable-length blocked sequential files, use the APPLY WRITE-ONLY clause for the file or use the AWO compiler option. This reduces the number of calls to Data Management Services to handle the I/Os. For performance considerations using the APPLY-WRITE-ONLY clause or the AWO compiler option, see “AWO” on page 17.

Using HFS files

You can process byte-stream HFS files as ORGANIZATIONAL SEQUENTIAL files using QSAM and specifying the PATH=fully-qualified-pathname and FILEDATA=BINARY options on the DD statement or using an environment variable to define the file.

You can process text HFS files as ORGANIZATION SEQUENTIAL files using QSAM and specifying the PATH=fully-qualified-pathname and FILEDATA=TEXT on the DD statement or as ORGANIZATION LINE SEQUENTIAL and specifying the PATH=fully-qualified-pathname on the DD statement.

Using VSAM files

When using VSAM files, increase the number of data buffers (BUFND) for sequential access or index buffers (BUFNI) for random access.

Also, select a control interval size (CISZ) that is appropriate for the application. A smaller CISZ results in faster retrieval for random processing at the expense of inserts, whereas a larger CISZ is more efficient for sequential processing. In general, using large CI and buffer space VSAM parameters may help to improve the performance of the application.

In general, sequential access is the most efficient, dynamic access the next, and random access is the least efficient. However, for relative record VSAM (ORGANIZATION IS RELATIVE), using ACCESS IS DYNAMIC when reading each record in a random order can be slower than using ACCESS IS RANDOM, since VSAM may prefetch multiple tracks of data when using ACCESS IS DYNAMIC. ACCESS IS DYNAMIC is optimal when reading one record in a random order and then reading several subsequent records sequentially.

Random access results in an increase in I/O activity because VSAM must access the index for each request. In order to give an idea of the differences in using SEQUENTIAL, RANDOM, and DYNAMIC access for sequential operations on an INDEXED file, we provide the measurements that were obtained from running a COBOL program that uses an ORGANIZATION IS INDEXED file on our test system; this may not be representative of the results on your system. The COBOL program does 10,000 writes and 10,000 reads. The ratios of CPU time, elapsed time and EXCP counts are shown, with ACCESS IS SEQUENTIAL used as the base line 100%.

Table 9. CPU time, elapsed time and EXCP counts with different access mode

Access mode	**CPU Time (seconds)**	**Elapsed (seconds)**	**EXCP counts**
ACCESS IS SEQUENTIAL	100%	100%	100%
ACCESS IS DYNAMIC with READ NEXT	134%	143%	193%
ACCESS IS DYNAMIC with READ	713%	1095%	7189%
ACCESS IS RANDOM	1405%	3140%	15190%

Note: For the DYNAMIC with READ and the RANDOM cases, the record key of the next sequential record was moved into the data buffer prior to the READ.

If you use alternate indexes, it is more efficient to use the Access Method Services to build them than to use the AIXBLD runtime option. Avoid using multiple alternate indexes when possible since updates will have to be applied through the primary paths and reflected through the multiple alternate paths.

Refer to Chapter 4.1 about the location of VSAM buffers.

To improve VSAM performance, you can use system-managed buffering (SMB) when possible. To use SMB, the data set must use System Management Subsystem (SMS) storage and be in Extended format (DSNTYPE=xxx in the data class, where xxx is some form of extended format). Then you can use one of the following, depending on the record access type needed:

1. AMP='ACCBIAS=DO': optimize for only random record access
2. AMP='ACCBIAS=SO': optimize for only sequential record access
3. AMP='ACCBIAS=DW': optimize for mainly random record access with some sequential access
4. AMP='ACCBIAS=SW': optimize for mainly sequential record access with some random access

Refer to *Tuning your program* in the *Enterprise COBOL for z/OS Programming Guide* for additional coding techniques and best practices.

Chapter 8. Program object size and PDSE requirement

For details, see the following topics:

Changes in load module size between V4 and V6

The most important reason for executable code growth between V4 and V6 is an advanced optimization in V6, which was introduced in V5, to inline some out-of-line PERFORM statements to their calling site. This inlining has a number of advantages for program performance. First, it avoids the overhead of dispatching and returning from the out-of-line perform. Second, it exposes the statements being performed to further optimization in the context of the surrounding statements. This latter reason often allows the optimizer, even at OPT(1), to exploit synergies and to eliminate redundancies in order to improve performance. Tuning has been done since V5.1 to reduce the number of PERFORMs that are inlined in cases where a performance increase is unlikely.

In V6, the INLINE and NOINLINE options are introduced to control whether the inlining of procedures (paragraphs or sections) referenced by PERFORM statements in the source program. For details, please view *INLINE* in the *Enterprise COBOL for z/OS Programming Guide*.

There are other reasons as well why the executable size may be larger in some cases compared to V4:

- Use of higher ARCH instructions that are usually 6 bytes versus 4 bytes for many lower arch instructions. For example:
  - Using more than one ARCH(8) move immediate instruction instead of one in memory move
  - Exploiting Decimal Floating Point for packed/zoned decimal arithmetic
- Various V6 optimizations over and above V4 results in more generated code but shorter path length and better performance. For example:
  - More advanced INSPECT inlining
  - Conditionally inlining some complex conversions
  - Conditionally correcting decimal precision for binary data
  - Speeding up MOVEs to numeric-edited and alpha-numeric-edited data items
- V4 used "base locator" pointers accessed by 4 byte load, but in V6, fewer base locators are used, but 6 byte long displacement instructions are used instead
- V6 has a higher unroll threshold than V4 when deciding to use multiple MVCs for large copies to avoid a more expensive MVCL instruction

Impact of TEST suboptions on program object size

As detailed in *TEST* in the *Enterprise COBOL for z/OS Programming Guide*, the TEST and NOTEST options have several suboptions in V6. The SOURCE/NOSOURCE, DWARF/NODWARF and SEPARATE/NOSEPARATE suboptions directly control if extra information used for debugging should or should not be included in the program object, and therefore can change the size of the object significantly.

The TEST option by itself and the suboption EJPD will also affect the program object size, but making it either greater or lesser, as these options may change the amount and types of optimizations performed by the compiler (so the resulting code and literal data areas may be smaller or larger).

Note that although the added debugging information will affect the size of the resulting program object, this will not affect the LOADed size.

Since the debugging information is in NOLOAD class segments, these parts of the program object are not loaded when the program is run, unless Debug Tool or Fault Analyzer or CEEDUMP processing explicitly requests it. So, unlike COBOL V4 and earlier versions, the size of the program object related to debugging information does not affect LOAD times or execution performance.

Since each suboption will impact the object size in a different way, let's examine each suboption separately. Results will be shown for OPTIMIZE(0) and OPTIMIZE(1) for each case, and all other options are kept at their default settings.

The size comparisons were gathered from a large selection of COBOL tests in our performance verification suite of applications.

First, let's compare the NOTEST suboption DWARF/NODWARF. The DWARF setting will cause basic DWARF diagnostic information to be included in the object.

Table 10. NOTEST(DWARF) % size increase over NOTEST(NODWARF)

Average Size	NOTEST(DWARF) % size increase over NOTEST(NODWARF)
OPTIMIZE(0)	96.1%
OPTIMIZE(1)	105.7%

So at both OPTIMIZE settings measured the overall object size roughly doubles when specifying DWARF overall the default NODWARF setting.

For TEST, let's first look at the option by itself versus NOTEST. The differences in object size in this case are due to a few reasons:

- The first major reason for the size increase is that TEST always causes full DWARF debugging information to be included in the object
- The second major reason for the size increase is that TEST by default enables the SOURCE suboption, so the generated DWARF debug information includes the expanded source code
- The third reason, and this generally matters less than the previous two reasons, is that TEST slightly inhibits optimization, and this may result in object size increases or decreases depending on the characteristics of the program

Table 11. TEST % size increase over NOTEST

Average Size	TEST % size increase over NOTEST
OPTIMIZE(0)	216.7%
OPTIMIZE(1)	237.8%

Next, let's look at the impact of the SOURCE/NOSOURCE suboption. This increase is directly related to the size of your expanded source file as it will be included in the DWARF debug information when TEST(SOURCE) is specified.

Despite the object size increase it causes, the advantage of specifying SOURCE is that since the DWARF information will contain the expanded source, a separate compiler listing will not be required by IBM Debug Tool.

Table 12. TEST(SOURCE) % size increase over TEST(NOSOURCE)	
Average Size	**TEST(SOURCE) % size increase over TEST(NOSOURCE)**
OPTIMIZE(0)	27.5%
OPTIMIZE(1)	27.0%

Finally, let's look at the impact to object size from toggling the TEST suboption EJPD/NOEJPD. This option does not change the amount or type of DWARF debug information included in the object, but only impacts the amount and types of optimizations performed by the compiler in order to meet the debugging requirements of EJPD.

Table 13. TEST(EJPD) % size increase over TEST(NOEJPD)	
Average Size	**TEST(EJPD) % size increase over TEST(NOEJPD)**
OPTIMIZE(0)	0%
OPTIMIZE(1)	4.0%

The 0% change at OPTIMIZE(0) makes sense, as this lowest level of optimization is already low enough to be not restricted by the extra debugging requirements of EJPD.

At OPTIMIZE(1), the more restrictive EJPD setting generally inhibits optimizations that would have resulted in smaller, and likely faster performing, executable code.

Why does COBOL V6 use PDSEs for executables?

As detailed in *Changes in compiling with Enterprise COBOL Version 5 and Version 6* in the *Enterprise COBOL for z/OS Migration Guide*, COBOL V6 executables must reside in a PDSE and can no longer be in a PDS.

This section describes some of the rationale for this change in behavior.

First, here is background information regarding PDS. When using PDS, customers reported problems in several areas:

- The need for frequent compressions
- Loss of data due to the directory being overwritten
- Performance impact due to a sequential directory search
- Performance delay if member added to beginning of directory
- When PDS went into multiple extents

In addition, PDS data sets cannot share update access to members without an enqueue on the entire dataset. More seriously though, a PDS library has to be taken down in order to perform compression to reclaim member space, or for a directory reallocation to reclaim wasted spaced (also known as gas).

Both of these can cause application downtime in production systems and are therefore very undesirable.

PDSEs, which were introduced in 1990, were designed to eliminate or at least reduce these problems and for the most part they have been successful. The initial rollout of PDSEs was rocky, and due to these problems long ago, many sites continue to avoid PDSEs to this day.

On the other hand, many other sites have moved their COBOL load libraries to PDSEs, and the process to do so is fairly mechanical. For example:

- Allocate new PDSE datasets with new names

- Copy Load Modules into PDSEs - these are converted to Program Objects
- Rename PDSs, then rename PDSEs

In fact, Enterprise COBOL has required program objects, therefore, PDSE for executables since 2001 for features such as long program names, object-oriented programs and for DLLs using the binder instead of the prelinker.

Only PDSEs (and z/OS USS files) can contain program objects, and this allows program management binder to solve some long standing existing problems using these program object features.

For example, once the 16M text size limit of load modules was hit, the only solution was an expensive redesign or refactoring of the program in order to make it smaller. With program objects the text size limit is increased to 1G.

This extra space also allows the COBOL compiler to perform more advanced optimizations that may increase program literal area, and ultimately object, size (with the goal of course of improving runtime performance). There are other advantages as well for COBOL using program objects:

- QY-con requires program objects
- Condition-sequential RLD support requires program objects (leading to a performance improvement for bootstrap invocation)
- Program objects can get page mapped 4K at a time for better performance
- Common reentrancy model with C/C++ requires program objects
- Looking into potential for the future XPLINK requires program objects and will be used for AMODE 64

A related issue is the different sharing rules across a SYSPLEX system. Unlike PDS libraries, PDSE data sets cannot be shared across SYSPLEX systems. Therefore, if existing pre-V6 PDS based COBOL load libraries are being shared, then V6 PDSE based load libraries can be moved using the following process:

- One SYSPLEX can be the writer/owner of master PDSE load library (development SYSPLEX)
- When PDSE load library is updated, push the new copy out to production SYSPLEX systems with XMIT or FTP
- The other SYSPLEX systems would then RECEIVE the updated PDSE load library

Appendix A. Using IBM Automatic Binary Optimizer for z/OS (ABO) to improve COBOL application performance

IBM Automatic Binary Optimizer for z/OS (ABO) enables you to improve the performance of already compiled IBM COBOL programs without the need for recompilation. The input to ABO is your compiled COBOL program modules, and it does not require source code, source code migration, or performance options tuning.

You can continue to use the latest version of Enterprise COBOL for new development, modernization, and maintenance, and if you don't have a recompile plan for some compiled programs or if some program source code is not available, you can use ABO to improve the performance of those COBOL modules.

To learn more about the relationship between COBOL and ABO, see Using ABO and Enterprise COBOL together in the *IBM Automatic Binary Optimizer for z/OS User's Guide*. To learn more about ABO, see the ABO product page.

Below are some of the frequently asked questions (FAQ) on ABO. More FAQs are available on the ABO FAQ page.

Is ABO a cost item?

The fully licensed and supported edition of ABO requires a license charge. Talk to your IBM sales representative or the online ABO sales to get the price.

ABO is also available as a 90-day cloud trial or on-premises trial, and both trial editions are no-charge. The ABO cloud trial does not require any installation, while the on-premises trial allows you to install ABO at your site.

What happens with ABO when I have the load module but not the source code?

ABO needs the load module only, and it does not look for the source. If you do not have the source, with the COBOL compiler you cannot recompile it because the compiler needs the source. However, ABO will work fine as long as the load module has been compiled with any COBOL compiler versions between VS COBOL II and Enterprise COBOL 6. For details, see Eligible compilers in the *IBM Automatic Binary Optimizer for z/OS User's Guide*.

When using ABO, should we save the original load module?

In case anything goes wrong, you can back up the original load module.

Appendix B. Intrinsic function implementation considerations

The COBOL intrinsic functions are implemented either by using LE callable services, library routines, inline code, or a combination of these. The following table shows how each of the intrinsic functions are implemented:

Table 14. Intrinsic Function Implementation

Function Name	LE Service	Library Routine	Inline Code
ACOS	X		
ANNUITY			X
ASIN	X		
ATAN	X		
CHAR			X
COS	X		
CURRENT-DATE	X		
DATE-OF-INTEGER	X		
DAY-OF-INTEGER	X		X
FACTORIAL			X
INTEGER			X
INTEGER-OF-DATE	X		
INTEGER-OF-DAY	X		X
INTEGER-PART			X
LENGTH			X
LOG	X		
LOG10	X		
LOWER-CASE			X
MAX			X
MEAN			X
MEDIAN		X	
MIDRANGE			X
MIN			X
MOD			X
NUMVAL		X	
NUMVAL-C		X	
ORD			X
ORD-MAX			X

Table 14. Intrinsic Function Implementation (continued)			
Function Name	**LE Service**	**Library Routine**	**Inline Code**
ORD-MIN			X
PRESENT-VALUE		X	
RANDOM	X		X
RANGE			X
REM (fixed-point)			X
REM (floating-point)	X		
REVERSE		X	
SIN	X		
SQRT	X		
STANDARD-DEVIATION		X	X
SUM			X
TAN	X		
UPPER-CASE		X	
VARIANCE		X	X
WHEN-COMPILED			X[1]
1. WHEN-COMPILED is a literal that is used whenever it is needed.			

Appendix C. Accessibility features for Enterprise COBOL for z/OS

Accessibility features assist users who have a disability, such as restricted mobility or limited vision, to use information technology content successfully. The accessibility features in z/OS provide accessibility for Enterprise COBOL for z/OS.

Accessibility features

z/OS includes the following major accessibility features:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- Ability to customize display attributes such as color, contrast, and font size

z/OS uses the latest W3C Standard, WAI-ARIA 1.0 (http://www.w3.org/TR/wai-aria/), to ensure compliance to US Section 508 (https://www.access-board.gov/ict/) and Web Content Accessibility Guidelines (WCAG) 2.0 (http://www.w3.org/TR/WCAG20/). To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Enterprise COBOL for z/OS online product documentation in IBM Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at http://www.ibm.com/support/knowledgecenter/en/about/releasenotes.html.

Keyboard navigation

Users can access z/OS user interfaces by using TSO/E or ISPF.

Users can also access z/OS services by using IBM Developer for z/OS.

For information about accessing these interfaces, see the following publications:

- *z/OS TSO/E Primer* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120)
- *z/OS TSO/E User's Guide* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3)
- *z/OS ISPF User's Guide Volume I* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70)
- IBM Developer for z/OS Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en)

These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Interface information

The Enterprise COBOL for z/OS online product documentation is available in IBM Knowledge Center, which is viewable from a standard web browser.

PDF files have limited accessibility support. With PDF documentation, you can use optional font enlargement, high-contrast display settings, and can navigate by keyboard alone.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.

Assistive technology products work with the user interfaces that are found in z/OS. For specific guidance information, see the documentation for the assistive technology product that you use to access z/OS interfaces.

Related accessibility information

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service
800-IBM-3383 (800-426-3383)
(within North America)

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see IBM Accessibility (www.ibm.com/able).

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan, Ltd. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software IBM Corporation 5 Technology Park Drive Westford, MA 01886 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated

through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1993, 2020.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.html.

Disclaimer

The performance considerations contained in this information were obtained by running sample programs in a particular hardware/software configuration using a selected set of tests and are presented as illustrations.

Since performance varies with configuration, program characteristics, and other installation and environment factors, results obtained in other operating environments may vary. We recommend that you construct sample programs representative of your workload and run your own experiments with a configuration applicable to your environment.

IBM does not represent, warrant, or guarantee that a user will achieve the same or similar results in the user's environment as the experimental results reported in this information.

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL*, and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module* and *ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL*
- *ANSI X3.172-2002, American National Standard Dictionary for Information Systems*
- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*
- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

American National Standard definitions are preceded by an asterisk (*).

A

*** abbreviated combined relation condition**
The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend
Abnormal termination of a program.

above the 16 MB line
Storage above the so-called 16 MB line (or boundary) but below the 2 GB bar. This storage is addressable only in 31-bit mode. Before IBM introduced the MVS/XA architecture in the 1980s, the virtual storage for a program was limited to 16 MB. Programs that have been compiled with a 24-bit mode can address only 16 MB of space, as though they were kept under an imaginary storage line. Since VS COBOL II, a program that has been compiled with a 31-bit mode can be above the 16 MB line.

*** access mode**
The manner in which records are to be operated upon within a file.

*** actual decimal point**
The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

actual document encoding
For an XML document, one of the following encoding categories that the XML parser determines by examining the first few bytes of the document:

- ASCII
- EBCDIC
- UTF-8
- UTF-16, either big-endian or little-endian
- Other unsupported encoding
- No recognizable encoding

*** alphabet-name**
A user-defined word, in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`, that assigns a name to a specific character set or collating sequence or both.

*** alphabetic character**
A letter or a space character.

alphanumeric character position
See *character position*.

alphabetic data item
A data item that is described with a PICTURE character string that contains only the symbol A. An alphabetic data item has USAGE DISPLAY.

*** alphanumeric character**
Any character in the single-byte character set of the computer.

alphanumeric data item
A general reference to a data item that is described implicitly or explicitly as USAGE DISPLAY, and that has category alphanumeric, alphanumeric-edited, or numeric-edited.

alphanumeric-edited data item
A data item that is described by a PICTURE character string that contains at least one instance of the symbol A or X and at least one of the simple insertion symbols B, 0, or /. An alphanumeric-edited data item has USAGE DISPLAY.

*** alphanumeric function**
A function whose value is composed of a string of one or more characters from the alphanumeric character set of the computer.

alphanumeric group item
A group item that is defined without a GROUP-USAGE NATIONAL clause. For operations such as INSPECT, STRING, and UNSTRING, an alphanumeric group item is processed as though all its content were described as USAGE DISPLAY regardless of the actual content of the group. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, or INITIALIZE, an alphanumeric group item is processed using group semantics.

alphanumeric literal
A literal that has an opening delimiter from the following set: ', ", X', X", Z', or Z". The string of characters can include any character in the character set of the computer.

*** alternate record key**
A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute)
An organization that consists of producers, consumers, and general-interest groups and establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument
(1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL or INVOKE statement, used for passing values to a called program or an invoked method.

*** arithmetic expression**
A numeric literal, an identifier representing a numeric elementary item, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**
The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**
A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction

Character	Meaning
*	Multiplication
/	Division
**	Exponentiation

*** arithmetic statement**
A statement that causes an arithmetic operation to be executed. The arithmetic statements are ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

array
An aggregate that consists of data objects, each of which can be uniquely referenced by subscripting. An array is roughly analogous to a COBOL table.

*** ascending key**
A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII
American National Standard Code for Information Interchange. The standard code uses a coded character set that is based on 7-bit coded characters (8 bits including parity check). The standard is used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

ASCII DBCS
See *double-byte ASCII*.

assignment-name
A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**
A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

AT END condition
A condition that is caused during the execution of a READ, RETURN, or SEARCH statement under certain conditions:

- A READ statement runs on a sequentially accessed file when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not available.
- A RETURN statement runs when no next logical record exists for the associated sort or merge file.
- A SEARCH statement runs when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

basic character set
The basic set of characters used in writing words, character-strings, and separators of the language. The basic character set is implemented in single-byte EBCDIC. The extended character set includes DBCS characters, which can be used in comments, literals, and user-defined words.

Synonymous with *COBOL character set* in the 85 COBOL Standard.

big-endian
The default format that the mainframe and the AIX® workstation use to store binary data and UTF-16 characters. In this format, the least significant byte of a binary data item is at the highest address and the least significant byte of a UTF-16 character is at the highest address. Compare with *little-endian*.

binary item
A numeric data item that is represented in binary notation (on the base 2 numbering system). The decimal equivalent consists of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search
A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

*** block**
A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical records that are either contained within the block or that overlap the block. Synonymous with *physical record*.

boolean condition
A boolean condition determines whether a boolean literal is true or false. A boolean condition can only be used in a constant conditional expression.

boolean literal
Can be either B'1', indicating a true value, or B'0', indicating a false value. Boolean literals can only be used in constant conditional expressions.

breakpoint
A place in a computer program, usually specified by an instruction, where external intervention or a monitor program can interrupt the program as it runs.

buffer
A portion of storage that is used to hold input or output data temporarily.

built-in function
See *intrinsic function*.

business method
A method of an enterprise bean that implements the business logic or rules of an application. (Oracle)

byte
A string that consists of a certain number of bits, usually eight, treated as a unit, and representing a character or a control function.

byte order mark (BOM)
A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big-endian or little-endian.

bytecode
Machine-independent code that is generated by the Java™ compiler and executed by the Java interpreter. (Oracle)

C

callable services
In Language Environment, a set of services that a COBOL program can invoke by using the conventional Language Environment-defined call interface. All programs that share the Language Environment conventions can use these services.

called program
A program that is the object of a CALL statement. At run time the called program and calling program are combined to produce a *run unit*.

*** calling program**
A program that executes a CALL to another program.

canonical decomposition
A way to represent a single precomposed Unicode character using two or more Unicode characters. A canonical decomposition is typically used to separate latin letters with a diacritical mark so that the latin letter and the diacritical mark are represented individually. See *precomposed character* for an example showing a precomposed Unicode character and its canonical decomposition.

case structure
A program-processing logic in which a series of conditions is tested in order to choose between a number of resulting actions.

cataloged procedure
A set of job control statements that are placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors in coding JCL.

CCSID
See *coded character set identifier*.

century window
A 100-year interval within which any two-digit year is unique. Several types of century window are available to COBOL programmers:

- For the windowing intrinsic functions `DATE-TO-YYYYMMDD`, `DAY-TO-YYYYDDD`, and `YEAR-TO-YYYY`, you specify the century window with *argument-2*.
- For Language Environment callable services, you specify the century window in CEESCEN.

*** character**
The basic indivisible unit of the language.

character encoding unit
A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For `USAGE NATIONAL`, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For `USAGE DISPLAY`, a character encoding unit corresponds to a byte.

For `USAGE DISPLAY-1`, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position
The amount of physical storage or presentation space required to hold or present one character. The term applies to any class of character. For specific classes of characters, the following terms apply:

- *Alphanumeric character position*, for characters represented in `USAGE DISPLAY`
- *DBCS character position*, for DBCS characters represented in `USAGE DISPLAY-1`
- *National character position*, for characters represented in `USAGE NATIONAL`; synonymous with *character encoding unit* for UTF-16

character set
A collection of elements that are used to represent textual information, but for which no coded representation is assumed. See also *coded character set*.

character string
A sequence of contiguous characters that form a COBOL word, a literal, a `PICTURE` character string, or a comment-entry. A character string must be delimited by separators.

checkpoint
A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

*** class**
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class. Classes can be defined hierarchically, allowing one class to inherit from another.

class (object-oriented)
The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.

*** class condition**
The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

*** class definition**
The COBOL source unit that defines a class.

class hierarchy
A tree-like structure that shows relationships among object classes. It places one class at the top and one or more layers of classes below it. Synonymous with *inheritance hierarchy*.

*** class identification entry**
An entry in the `CLASS-ID` paragraph of the `IDENTIFICATION DIVISION`; this entry contains clauses that specify the class-name and assign selected attributes to the class definition.

class-name (object-oriented)
The name of an object-oriented COBOL class definition.

*** class-name (of data)**
A user-defined word that is defined in the `SPECIAL-NAMES` paragraph of the `ENVIRONMENT DIVISION`; this word assigns a name to the proposition (for which a truth value can be defined) that the content of a data item consists exclusively of the characters that are listed in the definition of the class-name.

class object
The runtime object that represents a class.

*** clause**
An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

client
In object-oriented programming, a program or method that requests services from one or more methods in a class.

COBOL character set
The set of characters used in writing COBOL syntax. The complete COBOL character set consists of these characters:

Character	Meaning
0,1, . . . ,9	Digit
A,B, . . . ,Z	Uppercase letter
a,b, . . . ,z	Lowercase letter
	Space
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
$	Currency sign
,	Comma
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe

Character	Meaning
(	Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

*** COBOL word**
See *word*.

code page
An assignment of graphic characters and control function meanings to all code points. For example, one code page could assign characters and meanings to 256 code points for 8-bit code, and another code page could assign characters and meanings to 128 code points for 7-bit code. For example, one of the IBM code pages for English on the workstation is IBM-1252 and on the host is IBM-1047. A *coded character set*.

code point
A unique bit pattern that is defined in a coded character set (code page). Graphic symbols and control characters are assigned to code points.

coded character set
A set of unambiguous rules that establish a character set and the relationship between the characters of the set and their coded representation. Examples of coded character sets are the character sets as represented by ASCII or EBCDIC code pages or by the UTF-16 encoding scheme for Unicode.

coded character set identifier (CCSID)
An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

*** collating sequence**
The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

*** column**
A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

*** combined condition**
A condition that is the result of connecting two or more conditions with the AND or the OR logical operator. See also *condition* and *negated combined condition*.

combining characters
A Unicode character used to modify other succeeding or preceding Unicode characters. Combining characters are typically Unicode diacritical mark used to modify latin letters. See *precomposed character* for an example of combining character U+0308 (¨) used with latin letter U+0061 (a).

*** comment-entry**
An entry in the `IDENTIFICATION DIVISION` that is used for documentation and has no effect on execution.

comment line
A source program line represented by an asterisk (*) in the indicator area of the line or by an asterisk followed by greater-than sign (*>) as the first character string in the program text area (Area A plus Area B), and any characters from the character set of the computer that follow in Area A and Area B of that line. A comment line serves only for documentation. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in Area A and Area B of that line causes page ejection before the comment is printed.

*** common program**
A program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

*** compile**
(1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine-language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

compilation variable
A symbolic name for a particular literal value or the value of a compile-time arithmetic expression as specified by the DEFINE directive or by the DEFINE compiler option.

*** compile time**
The time at which COBOL source code is translated, by a COBOL compiler, to a COBOL object program.

compile-time arithmetic expression
A subset of arithmetic expressions that are specified in the DEFINE and EVALUATE directives or in a constant conditional expression. The difference between compile-time arithmetic expressions and regular arithmetic expressions is that in a compile-time arithmetic expression:

- The exponentiation operator shall not be specified.
- All operands shall be integer numeric literals or arithmetic expressions in which all operands are integer numeric literals.
- The expression shall be specified in such a way that a division by zero does not occur.

compiler
A program that translates source code written in a higher-level language into machine-language object code.

compiler-directing statement
A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are COPY, REPLACE, and USE.

*** complex condition**
A condition in which one or more logical operators act upon one or more conditions. See also *condition*, *negated simple condition*, and *negated combined condition*.

complex ODO
Certain forms of the OCCURS DEPENDING ON clause:

- Variably located item or group: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- Variably located table: A data item described by an OCCURS clause with the DEPENDING ON option is followed by a nonsubordinate data item described by an OCCURS clause.
- Table with variable-length elements: A data item described by an OCCURS clause contains a subordinate data item described by an OCCURS clause with the DEPENDING ON option.
- Index name for a table with variable-length elements.
- Element of a table with variable-length elements.

component
(1) A functional grouping of related files. (2) In object-oriented programming, a reusable object or program that performs a specific function and is designed to work with other components and applications. JavaBeans is Oracle's architecture for creating components.

composed form
Representation of a precomposed Unicode character through a canonical decomposition. See *precomposed character* for details.

*** computer-name**
A system-name that identifies the computer where the program is to be compiled or run.

condition (exception)
An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware or the operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition (expression)
A status of data at run time for which a truth value can be determined. Where used in this information in or in reference to "condition" (*condition-1*, *condition-2*,. . .) of a general format, the term refers to a conditional expression that consists of either a simple condition optionally parenthesized or a combined condition (consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses) for which a truth value can be determined. See also *simple condition*, *complex condition*, *negated simple condition*, *combined condition*, and *negated combined condition*.

*** conditional expression**
A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. See also *simple condition* and *complex condition*.

*** conditional phrase**
A phrase that specifies the action to be taken upon determination of the truth value of a condition that results from the execution of a conditional statement.

*** conditional statement**
A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program depends on this truth value.

*** conditional variable**
A data item one or more values of which has a condition-name assigned to it.

*** condition-name**
A user-defined word that assigns a name to a subset of values that a conditional variable can assume; or a user-defined word assigned to a status of an implementor-defined switch or device.

*** condition-name condition**
The proposition (for which a truth value can be determined) that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** CONFIGURATION SECTION**
A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

CONSOLE
A COBOL environment-name associated with the operator console.

constant conditional expression
A subset of conditional expressions that may be used in IF directives or WHEN phrases of the EVALUATE directives.

A constant conditional expression shall be one of the following items:

- A relation condition in which both operands are literals or arithmetic expressions that contain only literal terms. The condition shall follow the rules for relation conditions, with the following additions:
  - The operands shall be of the same category. An arithmetic expression is of the category numeric.
  - If literals are specified and they are not numeric literals, the relational operator shall be "IS EQUAL TO", "IS NOT EQUAL TO", "IS =", "IS NOT =", or "IS <>".

  See also *relation condition*.
- A defined condition. See also *defined condition*.
- A boolean condition. See also *boolean condition*.
- A complex condition formed by combining the above forms of simple conditions into complex conditions by using AND, OR, and NOT. Abbreviated combined relation conditions shall not be specified. See also *complex condition*.

contained program
A COBOL program that is nested within another COBOL program.

*** contiguous items**
Items that are described by consecutive entries in the DATA DIVISION, and that bear a definite hierarchic relationship to each other.

copybook
A file or library member that contains a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product. Synonymous with *copy file*.

*** counter**
A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cross-reference listing
The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

currency-sign value
A character string that identifies the monetary units stored in a numeric-edited item. Typical examples are $, USD, and EUR. A currency-sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency-sign value. See also *currency symbol*.

currency symbol
A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

*** current record**
In file processing, the record that is available in the record area associated with a file.

*** current volume pointer**
A conceptual entity that points to the current volume of a sequential file.

D

*** data clause**
A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

*** data description entry**
An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION
The division of a COBOL program or method that describes the data to be processed by the program or method: the files to be used and the records contained within them; internal WORKING-STORAGE records that will be needed; data to be made available in more than one program in the COBOL run unit.

*** data item**
A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

data set
Synonym for *file*.

*** data-name**
A user-defined word that names a data item described in a data description entry. When used in the general formats, data-name represents a word that must not be reference-modified, subscripted, or qualified unless specifically permitted by the rules for the format.

DBCS
See *double-byte character set (DBCS)*.

DBCS character
Any character defined in IBM's double-byte character set.

DBCS character position
See *character position*.

DBCS data item
A data item that is described by a PICTURE character string that contains at least one symbol G, or, when the NSYMBOL(DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has USAGE DISPLAY-1.

*** debugging line**
Any line with a D in the indicator area of the line.

*** debugging section**
A section that contains a USE FOR DEBUGGING statement.

*** declarative sentence**
A compiler-directing sentence that consists of a single USE statement terminated by the separator period.

*** declaratives**
A set of one or more special-purpose sections, written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word DECLARATIVE and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**
The logical removal of all editing characters from a numeric-edited data item in order to determine the unedited numeric value of the item.

defined condition
A compile-time condition that tests whether a compilation variable is defined. Defined conditions are specified in IF directives or WHEN phrases of the EVALUATE directives.

*** delimited scope statement**
Any statement that includes its explicit scope terminator.

*** delimiter**
A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

dependent region
In IMS, the MVS virtual storage region that contains message-driven programs, batch programs, or online utilities.

*** descending key**
A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit
Any of the numerals from 0 through 9. In COBOL, the term is not used to refer to any other symbol.

*** digit position**
The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**
The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

display floating-point data item
A data item that is described implicitly or explicitly as USAGE DISPLAY and that has a PICTURE character string that describes an external floating-point data item.

*** division**
A collection of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

*** division header**
A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

DLL
See *dynamic link library (DLL)*.

DLL application
An application that references imported programs, functions, or variables.

DLL linkage
A CALL in a program that has been compiled with the DLL and NODYNAM options; the CALL resolves to an exported name in a separate module, or to an INVOKE of a method that is defined in a separate module.

do construct
In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an inline PERFORM statement functions in the same way.

do-until
In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

do-while
In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

document type declaration
An XML element that contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.

document type definition (DTD)
The grammar for a class of XML documents. See *document type declaration*.

double-byte ASCII
An IBM character set that includes DBCS and single-byte ASCII characters. (Also known as ASCII DBCS.)

double-byte EBCDIC
An IBM character set that includes DBCS and single-byte EBCDIC characters. (Also known as EBCDIC DBCS.)

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points,

require double-byte character sets. Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

DWARF
DWARF was developed by the UNIX International Programming Languages Special Interest Group (SIG). It is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by supplying language-independent debugging information. A DWARF file contains debugging data organized into different elements. For more information, see *DWARF program information* in the *DWARF/ELF Extensions Library Reference*.

*** dynamic access**
An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

dynamic CALL
A CALL *literal* statement in a program that has been compiled with the DYNAM option and the NODLL option, or a CALL *identifier* statement in a program that has been compiled with the NODLL option.

dynamic link library (DLL)
A file that contains executable code and data that are bound to a program at load time or run time, rather than during linking. Several applications can share the code and data in a DLL simultaneously. Although a DLL is not part of the executable file for a program, it can be required for an executable file to run properly.

dynamic storage area (DSA)
Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). A DSA is allocated upon invocation of a program or function and persists for the duration of the invocation instance. DSAs are generally allocated within stack segments managed by Language Environment.

E

*** EBCDIC (Extended Binary-Coded Decimal Interchange Code)**
A coded character set based on 8-bit coded characters.

EBCDIC character
Any one of the symbols included in the EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

EBCDIC DBCS
See *double-byte EBCDIC*.

edited data item
A data item that has been modified by suppressing zeros or inserting editing characters or both.

*** editing character**
A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
$	Currency sign

Character	Meaning
,	Comma (decimal point)
.	Period (decimal point)
/	Slant (forward slash)

EGCS
See *extended graphic character set (EGCS)*.

EJB
See *Enterprise JavaBeans*.

EJB container
A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB or J2EE server. (Oracle)

EJB server
Software that provides services to an EJB container. An EJB server can host one or more EJB containers. (Oracle)

element (text element)
One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

*** elementary item**
A data item that is described as not being further logically subdivided.

encapsulation
In object-oriented programming, the technique that is used to hide the inherent details of an object. The object provides an interface that queries and manipulates the data without exposing its underlying structure. Synonymous with *information hiding*.

enclave
When running under Language Environment, an enclave is analogous to a run unit. An enclave can create other enclaves by using `LINK` and by using the system() function in C.

encoding unit
See *character encoding unit*.

end class marker
A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

```
END CLASS class-name.
```

end method marker
A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:

```
END METHOD method-name.
```

*** end of `PROCEDURE DIVISION`**
The physical position of a COBOL source program after which no further procedures appear.

*** end program marker**
A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

```
END PROGRAM program-name.
```

enterprise bean
A component that implements a business task and resides in an EJB container. (Oracle)

Enterprise JavaBeans
A component architecture defined by Oracle for the development and deployment of object-oriented, distributed, enterprise-level applications.

*** entry**
Any descriptive set of consecutive clauses terminated by a separator period and written in the `IDENTIFICATION DIVISION`, `ENVIRONMENT DIVISION`, or `DATA DIVISION` of a COBOL program.

*** environment clause**
A clause that appears as part of an `ENVIRONMENT DIVISION` entry.

`ENVIRONMENT DIVISION`
One of the four main component parts of a COBOL program, class definition, or method definition. The `ENVIRONMENT DIVISION` describes the computers where the source program is compiled and those where the object program is run. It provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

environment-name
A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, program switches or all of these. When an environment-name is associated with a mnemonic-name in the `ENVIRONMENT DIVISION`, the mnemonic-name can be substituted in any format in which such substitution is valid.

environment variable
Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

escape sequence
A sequence of characters that are used to represent certain special characters within string literals and character literals.

Escape sequences consist of two or more characters, the first of which is the backslash (\) character, which is called the "escape character"; the remaining characters determine the interpretation of the escape sequence. For example, `\n` is an escape sequence that denotes a newline character.

Escape sequences are used in programming languages such as C, C++, Java, or Python. COBOL does not have the concept of "escape sequence" or "escape character". To handle special characters within COBOL literals, see *Basic alphanumeric literals* and *DBCS literals* in the *Enterprise COBOL for z/OS Language Reference*.

execution time
See *run time*.

execution-time environment
See *runtime environment*.

*** explicit scope terminator**
A reserved word that terminates the scope of a particular `PROCEDURE DIVISION` statement.

exponent
A number that indicates the power to which another number (the base) is to be raised. Positive exponents denote multiplication; negative exponents denote division; and fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ** followed by the exponent.

*** expression**
An arithmetic or conditional expression.

*** extend mode**
The state of a file after execution of an `OPEN` statement, with the `EXTEND` phrase specified for that file, and before the execution of a `CLOSE` statement, without the `REEL` or `UNIT` phrase for that file.

extended graphic character set (EGCS)
A graphic character set, such as a kanji character set, that requires two bytes to identify each graphic character. It is refined and replaced by *double-byte character set (DBCS)*.

Extensible Markup Language
See *XML*.

extensions
COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

external code page
For XML documents, the value specified by the CODEPAGE compiler option.

*** external data**
The data that is described in a program as external data items and external file connectors.

*** external data item**
A data item that is described as part of an external record in one or more programs of a run unit and that can be referenced from any program in which it is described.

*** external data record**
A logical record that is described in one or more programs of a run unit and whose constituent data items can be referenced from any program in which they are described.

external decimal data item
See *zoned decimal data item* and *national decimal data item*.

*** external file connector**
A file connector that is accessible to one or more object programs in the run unit.

external floating-point data item
See *display floating-point data item* and *national floating-point data item*.

external program
The outermost program. A program that is not nested.

*** external switch**
A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

factory data
Data that is allocated once for a class and shared by all instances of the class. Factory data is declared in the WORKING-STORAGE SECTION of the DATA DIVISION in the FACTORY paragraph of the class definition, and is equivalent to Java private static data.

factory method
A method that is supported by a class independently of an object instance. Factory methods are declared in the FACTORY paragraph of the class definition, and are equivalent to Java public static methods. They are typically used to customize the creation of objects.

*** figurative constant**
A compiler-generated value referenced through the use of certain reserved words.

*** file**
A collection of logical records.

*** file attribute conflict condition**
An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file clause**
A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

*** file connector**
A storage area that contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File-Control
The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

file control block
Block containing the addresses of I/O routines, information about how they were opened and closed, and a pointer to the file information block.

*** file control entry**
A SELECT clause and all its subordinate clauses that declare the relevant physical attributes of a file.

FILE-CONTROL paragraph
A paragraph in the ENVIRONMENT DIVISION in which the data files for a given source unit are declared.

*** file description entry**
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

*** file-name**
A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the FILE SECTION of the DATA DIVISION.

*** file organization**
The permanent logical file structure established at the time that a file is created.

file position indicator
A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not available, or that the AT END condition already exists, or that no valid next record has been established.

*** FILE SECTION**
The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

file system
The collection of files that conform to a specific set of data-record and file-description protocols, and a set of programs that manage these files.

*** fixed file attributes**
Information about a file that is established when a file is created and that cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed-length record**
A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of bytes.

fixed-point item
A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format can be either binary, packed decimal, or external decimal.

floating comment indicators (*>)
A floating comment indicator indicates a comment line if it is the first character string in the program-text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program-text area.

floating point
A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral) and a value obtained by raising the implicit floating-point base to a power denoted

by the exponent (the second numeral). For example, a floating-point representation of the number 0.0001234 is 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

floating-point data item
A numeric data item that contains a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power that the exponent specifies.

*** format**
A specific arrangement of a set of data.

*** function**
A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**
A syntactically correct combination of character strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name
A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

function-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS FUNCTION-POINTER clause contains the address of a function entry point. Typically used to communicate with C and Java programs.

G

garbage collection
The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

*** global name**
A name that is declared in only one program but that can be referenced from the program and from any program contained within the program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

global reference
A reference to an object that is outside the scope of a method.

group item
(1) A data item that is composed of subordinate data items. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator
A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

H

header label
(1) A data-set label that precedes the data records in a unit of recording media. (2) Synonym for *beginning-of-file label*.

hide (a method)
To redefine (in a subclass) a factory or static method defined with the same method-name in a parent class. Thus, the method in the subclass *hides* the method in the parent class.

*** high-order end**
The leftmost character of a string of characters.

hiperspace
In a z/OS environment, a range of up to 2 GB of contiguous virtual storage addresses that a program can use as a buffer.

I

IBM COBOL extension
COBOL syntax and semantics supported by IBM compilers in addition to those described in the 85 COBOL Standard.

IDENTIFICATION DIVISION
One of the four main component parts of a COBOL program, class definition, or method definition. The `IDENTIFICATION DIVISION` identifies the program, class, or method. The `IDENTIFICATION DIVISION` can include the following documentation: author name, installation, or date.

*** identifier**
A syntactically correct combination of character strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item that is a function, a function-identifier is used.

IGZCBSN
The bootstrap routine for COBOL/370 Release 1. It must be link-edited with any module that contains a COBOL/370 Release 1 program.

IGZCBSO
The bootstrap routine for COBOL for MVS & VM Release 2, COBOL for OS/390 & VM and Enterprise COBOL. It must be link-edited with any module that contains a COBOL for MVS & VM Release 2, COBOL for OS/390 & VM or Enterprise COBOL program.

IGZEBST
The bootstrap routine for VS COBOL II. It must be link-edited with any module that contains a VS COBOL II program.

ILC
InterLanguage Communication. Interlanguage communication is defined as programs that call or are called by other high-level languages. Assembler is not considered a high-level language; thus, calls to and from assembler programs are not considered ILC.

*** imperative statement**
A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement can consist of a sequence of imperative statements.

*** implicit scope terminator**
A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

IMS
Information Management System, IBM licensed product. IMS supports hierarchical databases, data communication, translation processing, and database backout and recovery.

*** index**
A computer storage area or register, the content of which represents the identification of a particular element in a table.

*** index data item**
A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name
An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**
A file with indexed organization.

*** indexed organization**
The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing
Synonymous with *subscripting* using index-names.

*** index-name**
A user-defined word that names an index associated with a specific table.

inheritance
A mechanism for using the implementation of a class as the basis for another class. By definition, the inheriting class conforms to the inherited classes. Enterprise COBOL does not support *multiple inheritance*; a subclass has exactly one immediate superclass.

inheritance hierarchy
See *class hierarchy*.

*** initial program**
A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**
The state of a program when it is first called in a run unit.

inline
In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

inline comments
An inline comment is identified by a floating comment indicator (*>) preceded by one or more character-strings in the program-text area, and can be written on any line of a compilation group. All characters that follow the floating comment indicator up to the end of area B are comment text.

*** input file**
A file that is opened in the input mode.

*** input mode**
The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** input-output file**
A file that is opened in the I-O mode.

*** INPUT-OUTPUT SECTION**
The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data at run time.

*** input-output statement**
A statement that causes files to be processed by performing operations on individual records or on the file as a unit. The input-output statements are ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**
A set of statements, to which control is given during the execution of a format 1 SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data
Data that defines the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION in the OBJECT paragraph of the class definition. The state of an object also includes the state of the instance variables introduced by classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

*** integer**
(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the

right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

integer function
A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

Interactive System Productivity Facility (ISPF)
An IBM software product that provides a menu-driven interface for the TSO or VM user. ISPF includes library utilities, a powerful editor, and dialog management.

interlanguage communication (ILC)
The ability of routines written in different programming languages to communicate. ILC support lets you readily build applications from component routines written in a variety of languages.

intermediate result
An intermediate field that contains the results of a succession of arithmetic operations.

*** internal data**
The data that is described in a program and excludes all external data items and external file connectors. Items described in the `LINKAGE SECTION` of a program are treated as internal data.

*** internal data item**
A data item that is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item
A data item that is described as `USAGE PACKED-DECIMAL` or `USAGE COMP-3`, and that has a `PICTURE` character string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed-decimal data item*.

*** internal file connector**
A file connector that is accessible to only one object program in the run unit.

internal floating-point data item
A data item that is described as `USAGE COMP-1` or `USAGE COMP-2`. `COMP-1` defines a single-precision floating-point data item. `COMP-2` defines a double-precision floating-point data item. There is no `PICTURE` clause associated with an internal floating-point data item.

*** intrarecord data structure**
The entire collection of groups and elementary data items from a logical record that a contiguous subset of the data description entries defines. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

intrinsic function
A predefined function, such as a commonly used arithmetic function, called by a built-in function reference.

*** invalid key condition**
A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be not valid.

*** `I-O-CONTROL`**
The name of an `ENVIRONMENT DIVISION` paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

*** `I-O-CONTROL` entry**
An entry in the `I-O-CONTROL` paragraph of the `ENVIRONMENT DIVISION`; this entry contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

*** I-O mode**
The state of a file after execution of an `OPEN` statement, with the `I-O` phrase specified, for that file and before the execution of a `CLOSE` statement without the `REEL` or `UNIT` phase for that file.

*** I-O status**
A conceptual entity that contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the `FILE STATUS` clause in the file control entry for the file.

is-a
A relationship that characterizes classes and subclasses in an inheritance hierarchy. Subclasses that have an is-a relationship to a class inherit from that class.

ISPF
See *Interactive System Productivity Facility (ISPF)*.

iteration structure
A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

J

J2EE
See *Java 2 Platform, Enterprise Edition (J2EE)*.

Java 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications, defined by Oracle. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Oracle)

Java Batch Launcher and Toolkit for z/OS (JZOS)
A set of tools that helps you develop z/OS Java applications that run in a traditional batch environment, and that access z/OS system services.

Java batch-processing program (JBP)
An IMS batch-processing program that has access to online databases and output message queues. JBPs run online, but like programs in a batch environment, they are started with JCL or in a TSO session.

Java batch-processing region
An IMS dependent region in which only Java batch-processing programs are scheduled.

Java Database Connectivity (JDBC)
A specification from Oracle that defines an API that enables Java programs to access databases.

Java message-processing program (JMP)
A Java application program that is driven by transactions and has access to online IMS databases and message queues.

Java message-processing region
An IMS dependent region in which only Java message-processing programs are scheduled.

Java Native Interface (JNI)
A programming interface that lets Java code that runs inside a Java virtual machine (JVM) interoperate with applications and libraries written in other programming languages.

Java virtual machine (JVM)
A software implementation of a central processing unit that runs compiled Java programs.

JavaBeans
A portable, platform-independent, reusable component model. (Oracle)

JBP
See *Java batch-processing program (JBP)*.

JDBC
See *Java Database Connectivity (JDBC)*.

JMP
See *Java message-processing program (JMP)*.

job control language (JCL)
A control language used to identify a job to an operating system and to describe the job's requirements.

JSON
JSON (JavaScript Object Notation) is a lightweight data-interchange format.

JVM
See *Java virtual machine (JVM)*.

JZOS
See *Java Batch Launcher and Toolkit for z/OS*.

K

K
When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key**
A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

*** key of reference**
The key, either prime or alternate, currently being used to access records within an indexed file.

*** keyword**
A context-sensitive word or a reserved word whose presence is required when the format in which the word appears is used in a source unit.

kilobyte (KB)
One kilobyte equals 1024 bytes.

L

*** language-name**
A system-name that specifies a particular programming language.

Language Environment
Short form of z/OS Language Environment. A set of architectural constructs and interfaces that provides a common runtime environment and runtime services for C, C++, COBOL, FORTRAN and PL/I applications. It is required for programs compiled by Language Environment-conforming compilers and for Java applications.

Language Environment-conforming
A characteristic of compiler products (such as Enterprise COBOL, COBOL for OS/390 & VM, COBOL for MVS & VM, C/C++ for MVS & VM, PL/I for MVS & VM) that produce object code conforming to the Language Environment conventions.

last-used state
A state that a program is in if its internal values remain the same as when the program was exited (the values are not reset to their initial values).

*** letter**
A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

*** level indicator**
Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the `DATA DIVISION` are: `CD`, `FD`, and `SD`.

*** level-number**
A user-defined word (expressed as a two-digit number) that indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

*** library-name**
A user-defined word that names a COBOL library that the compiler is to use for compiling a given source program.

*** library text**
A sequence of text words, comment lines, inline comments, the separator space, or the separator pseudo-text delimiter in a COBOL library.

Lilian date
The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

*** linage-counter**
A special register whose value points to the current position within the page body.

link
(1) The combination of the link connection (the transmission medium) and two link stations, one at each end of the link connection. A link can be shared among multiple links in a multipoint or token-ring configuration. (2) To interconnect items of data or portions of one or more computer programs; for example, linking object programs by a linkage-editor to produce an executable file.

LINKAGE SECTION
The section in the DATA DIVISION of the called program or invoked method that describes data items available from the calling program or invoking method. Both the calling program or invoking method and the called program or invoked method can refer to these data items.

linker
A term that refers to either the z/OS binder (linkage-editor).

literal
A character string whose value is specified either by the ordered set of characters comprising the string or by the use of a figurative constant.

little-endian
The default format that Intel processors use to store binary data and UTF-16 characters. In this format, the most significant byte of a binary data item is at the highest address and the most significant byte of a UTF-16 character is at the highest address. Compare with *big-endian*.

local reference
A reference to an object that is within the scope of your method.

locale
A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as character code page, collating sequence, date and time format, monetary value representation, numeric value representation, or language.

*** LOCAL-STORAGE SECTION**
The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in the VALUE clauses.

*** logical operator**
One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

*** logical record**
The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. Synonymous with *record*.

*** low-order end**
The rightmost character of a string of characters.

M

main program
In a hierarchy of programs and subroutines, the first program that receives control when the programs are run within a process.

makefile
A text file that contains a list of the files for your application. The make utility uses this file to update the target files with the latest changes.

*** mass storage**
A storage medium in which data can be organized and maintained in both a sequential manner and a nonsequential manner.

*** mass storage device**
A device that has a large storage capacity, such as a magnetic disk.

*** mass storage file**
A collection of records that is stored in a mass storage medium.

*** megabyte (MB)**
One megabyte equals 1,048,576 bytes.

*** merge file**
A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

message-processing program (MPP)
An IMS application program that is driven by transactions and has access to online IMS databases and message queues.

message queue
The data set on which messages are queued before being processed by an application program or sent to a terminal.

method
Procedural code that defines an operation supported by an object and that is executed by an INVOKE statement on that object.

*** method definition**
The COBOL source code that defines a method.

*** method identification entry**
An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION; this entry contains a clause that specifies the method-name.

method invocation
A communication from one object to another that requests the receiving object to execute a method.

method-name
The name of an object-oriented operation. When used to invoke the method, the name can be an alphanumeric or national literal or a category alphanumeric or category national data item. When used in the METHOD-ID paragraph to define the method, the name must be an alphanumeric or national literal.

method hiding
See *hide*.

method overloading
See *overload*.

method overriding
See *override*.

*** mnemonic-name**
A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

module definition file
A file that describes the code segments within a program object.

MPP
See *message-processing program (MPP)*.

multitasking
A mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks.

multithreading
Concurrent operation of more than one path of execution within a computer. Synonymous with *multiprocessing*.

N

name
A word (composed of not more than 30 characters) that defines a COBOL operand.

namespace
See *XML namespace*.

national character
(1) A UTF-16 character in a USAGE NATIONAL data item or national literal. (2) Any character represented in UTF-16.

national character data
A general reference to data represented in UTF-16.

national character position
See *character position*.

national data
See *national character data*.

national data item
A data item of category national, national-edited, or numeric-edited of USAGE NATIONAL.

national decimal data item
An external decimal data item that is described implicitly or explicitly as USAGE NATIONAL and that contains a valid combination of PICTURE symbols 9, S, P, and V.

national-edited data item
A data item that is described by a PICTURE character string that contains at least one instance of the symbol N and at least one of the simple insertion symbols B, 0, or /. A national-edited data item has USAGE NATIONAL.

national floating-point data item
An external floating-point data item that is described implicitly or explicitly as USAGE NATIONAL and that has a PICTURE character string that describes a floating-point data item.

national group item
A group item that is explicitly or implicitly described with a GROUP-USAGE NATIONAL clause. A national group item is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This processing ensures correct padding and truncation of national characters, as contrasted with defining USAGE NATIONAL data items within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE, a national group is processed using group semantics.

*** native character set**
The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**
The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

native method
A Java method with an implementation that is written in another programming language, such as COBOL.

*** negated combined condition**
The NOT logical operator immediately followed by a parenthesized combined condition. See also *condition* and *combined condition*.

*** negated simple condition**
The NOT logical operator immediately followed by a simple condition. See also *condition* and *simple condition*.

nested program
A program that is directly contained within another program.

*** next executable sentence**
The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**
The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**
The record that logically follows the current record of a file.

*** noncontiguous items**
Elementary data items in the `WORKING-STORAGE SECTION` and `LINKAGE SECTION` that bear no hierarchic relationship to other data items.

*** noncontiguous items**
Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONs that bear no hierarchic relationship to other data items.

*** nonnumeric item**
A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

null
A figurative constant that is used to assign, to pointer data items, the value of an address that is not valid. `NULLS` can be used wherever `NULL` can be used.

*** numeric character**
A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric data item
(1) A data item whose description restricts its content to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a +, -, or other representation of an operational sign. (2) A data item of category numeric, internal floating-point, or external floating-point. A numeric data item can have `USAGE DISPLAY`, `NATIONAL`, `PACKED-DECIMAL`, `BINARY`, `COMP`, `COMP-1`, `COMP-2`, `COMP-3`, `COMP-4`, or `COMP-5`.

numeric-edited data item
A data item that contains numeric data in a form suitable for use in printed output. The data item can consist of external decimal digits from 0 through 9, the decimal separator, commas, the currency sign, sign control characters, and other editing characters. A numeric-edited item can be represented in either `USAGE DISPLAY` or `USAGE NATIONAL`.

*** numeric function**
A function whose class and category are numeric but that for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric item**
A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

*** numeric literal**
A literal composed of one or more numeric characters that can contain a decimal point or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

object
An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior. Each object in the class is said to be an instance of the class.

object code
Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** OBJECT-COMPUTER**
The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the object program is run, is described.

*** object computer entry**
An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the object program is to be executed.

object deck
A portion of an object program suitable as input to a linkage-editor. Synonymous with *object module* and *text deck*.

object instance
A single object, of possibly many, instantiated from the specifications in the object paragraph of a COBOL class definition. An object instance has a copy of all the data described in its class definition and all inherited data. The methods associated with an object instance includes the methods defined in its class definition and all inherited methods.

An object instance can be an instance of a Java class.

object module
Synonym for *object deck* or *text deck*.

*** object of entry**
A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program, that immediately follows the subject of the entry.

object-oriented programming
A programming approach based on the concepts of encapsulation and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on the data objects that comprise the problem and how they are manipulated, not on how something is accomplished.

object program
A set or group of executable machine-language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program or class definition. Where there is no danger of ambiguity, the word *program* can be used in place of *object program*.

object reference
A value that identifies an instance of a class. If the class is not specified, the object reference is universal and can apply to instances of any class.

*** object time**
The time at which an object program is executed. Synonymous with *run time*.

*** obsolete element**
A COBOL language element in the 85 COBOL Standard that was deleted from the 2002 COBOL Standard.

ODO object
In the example below, X is the object of the OCCURS DEPENDING ON clause (ODO object).

```
WORKING-STORAGE SECTION.
01  TABLE-1.
    05  X                   PIC S9.
    05  Y OCCURS 3 TIMES
          DEPENDING ON X    PIC X.
```

The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject
In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

*** open mode**
The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

*** operand**
(1) The general definition of operand is "the component that is operated upon." (2) For the purposes of this document, any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operation
A service that can be requested of an object.

*** operational sign**
An algebraic sign that is associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file
A file that is declared as being not necessarily available each time the object program is run.

*** optional word**
A reserved word that is included in a specific format only to improve the readability of the language. Its presence is optional to the user when the format in which the word appears is used in a source unit.

*** output file**
A file that is opened in either output mode or extend mode.

*** output mode**
The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**
A set of statements to which control is given during execution of a format 1 SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition
A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overload
To define a method with the same name as another method that is available in the same class, but with a different signature. See also *signature*.

override
To redefine an instance method (inherited from a parent class) in a subclass.

P

package
A group of related Java classes, which can be imported individually or as a whole.

packed-decimal data item
See *internal decimal data item*.

padding character
An alphanumeric or national character that is used to fill the unused character positions in a physical record.

page
A vertical division of output data that represents a physical separation of the data. The separation is based on internal logical requirements or external characteristics of the output medium or both.

*** page body**
That part of the logical page in which lines can be written or spaced or both.

*** paragraph**
In the PROCEDURE DIVISION, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION, a paragraph header followed by zero, one, or more entries.

*** paragraph header**
A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

```
PROGRAM-ID. (Program IDENTIFICATION
   DIVISION)
CLASS-ID. (Class IDENTIFICATION DIVISION)
METHOD-ID. (Method IDENTIFICATION
   DIVISION)
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
REPOSITORY. (Program or Class
   CONFIGURATION SECTION)
FILE-CONTROL.
I-O-CONTROL.
```

*** paragraph-name**
A user-defined word that identifies and begins a paragraph in the PROCEDURE DIVISION.

parameter
(1) Data passed between a calling program and a called program. (2) A data element in the USING phrase of a method invocation. Arguments provide additional information that the invoked method can use to perform the requested operation.

Persistent Reusable JVM
A JVM that can be serially reused for transaction processing by resetting the JVM between transactions. The reset phase restores the JVM to a known initialization state.

*** phrase**
An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**
See *block*.

pointer data item
A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

port
(1) To modify a computer program to enable it to run on a different platform. (2) In the Internet suite of protocols, a specific logical connector between the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and a higher-level protocol or application. A port is identified by a port number.

portability
The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

precomposed character
A single Unicode character that can be represented using two or more Unicode characters through a canonical decomposition. A precomposed character does not have the same physical representation as its composed character form. For example, Unicode character U+00E4 (ä) is a precomposed character that can be represented as a combination of Unicode characters U+0061 + U+0308 (ä) - latin small letter a + combining diaeresis. A precomposed character is typically used to represent a latin letter with a diacritical mark or some other combining character.

preinitialization
The initialization of the COBOL runtime environment in preparation for multiple calls from programs, especially non-COBOL programs. The environment is not terminated until an explicit termination.

*** prime record key**
A key whose contents uniquely identify a record within an indexed file.

*** priority-number**
A user-defined word that classifies sections in the PROCEDURE DIVISION for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as either one or two digits.

private
As applied to factory data or instance data, accessible only by methods of the class that defines the data.

*** procedure**
A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the PROCEDURE DIVISION.

*** procedure branching statement**
A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source code. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase), XML PARSE.

PROCEDURE DIVISION
The COBOL division that contains instructions for solving a problem.

procedure integration
One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a call to a contained program is replaced by the program code.

*** procedure-name**
A user-defined word that is used to name a paragraph or section in the PROCEDURE DIVISION. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure pointer
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

procedure-pointer data item
A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point. Typically used to communicate with COBOL and Language Environment programs.

process
The course of events that occurs during the execution of all or part of a program. Multiple processes can run concurrently, and programs that run within a process can share resources.

program
(1) A sequence of instructions suitable for processing by a computer. Processing may include the use of a compiler to prepare the program for execution, as well as a runtime environment to execute it. (2)

A logical assembly of one or more interrelated modules. Multiple copies of the same program can be run in different processes.

program-name
In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or an alphanumeric literal that identifies a COBOL source program.

*** program identification entry**
In the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION, an entry that contains clauses that specify the program-name and assign selected program attributes to the program.

program-name
In the IDENTIFICATION DIVISION and the end program marker, a user-defined word or alphanumeric literal that identifies a COBOL source program.

project
The complete set of data and actions that are required to build a target, such as a dynamic link library (DLL) or other executable (EXE).

*** pseudo-text**
A sequence of text words, comment lines, inline comments, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**
Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**
A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

Q

QSAM (Queued Sequential Access Method)
An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

*** qualified data-name**
An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**
(1) A data-name or a name associated with a level indicator that is used in a reference either together with another data-name (which is the name of an item that is subordinate to the qualifier) or together with a condition-name. (2) A section-name that is used in a reference together with a paragraph-name specified in that section. (3) A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**
An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*** record**
See *logical record*.

*** record area**
A storage area allocated for the purpose of processing the record described in a record description entry in the FILE SECTION of the DATA DIVISION. In the FILE SECTION, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

*** record description**
See *record description entry*.

*** record description entry**
The total set of data description entries associated with a particular record. Synonymous with *record description*.

recording mode
The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

record key
A key whose contents identify a record within an indexed file.

*** record-name**
A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

*** record number**
The ordinal number of a record in the file whose organization is sequential.

recording mode
The format of the logical records in a file. Recording mode can be F (fixed length), V (variable length), S (spanned), or U (undefined).

recursion
A program calling itself or being directly or indirectly called by one of its called programs.

recursively capable
A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

reel
A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. Synonymous with *unit* and *volume*.

reentrant
The attribute of a program or routine that lets more than one user share a single copy of a program object.

*** reference format**
A format that provides a standard method for describing COBOL source programs.

reference modification
A method of defining a new category alphanumeric, category DBCS, or category national data item by specifying the leftmost character and length relative to the leftmost character position of a USAGE DISPLAY, DISPLAY-1, or NATIONAL data item.

*** reference-modifier**
A syntactically correct combination of character strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**
See *relational operator* or *relation condition*.

*** relation character**
A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

*** relation condition**
The proposition (for which a truth value can be determined) that the value of an arithmetic expression, data item, alphanumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, alphanumeric literal, or index name. See also *relational operator*.

*** relational operator**
A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	Meaning
`IS GREATER THAN`	Greater than
`IS >`	Greater than
`IS NOT GREATER THAN`	Not greater than
`IS NOT >`	Not greater than
`IS LESS THAN`	Less than
`IS <`	Less than
`IS NOT LESS THAN`	Not less than
`IS NOT <`	Not less than
`IS EQUAL TO`	Equal to
`IS =`	Equal to
`IS NOT EQUAL TO`	Not equal to
`IS NOT =`	Not equal to
`IS GREATER THAN OR EQUAL TO`	Greater than or equal to
`IS >=`	Greater than or equal to
`IS LESS THAN OR EQUAL TO`	Less than or equal to
`IS <=`	Less than or equal to

*** relative file**
A file with relative organization.

*** relative key**
A key whose contents identify a logical record in a relative file.

*** relative organization**
The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the logical ordinal position of the record in the file.

*** relative record number**
The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

*** reserved word**
A COBOL word that is specified in the list of words that can be used in a COBOL source program, but that must not appear in the program as a user-defined word or system-name.

*** resource**
A facility or service, controlled by the operating system, that an executing program can use.

*** resultant identifier**
A user-defined data item that is to contain the result of an arithmetic operation.

reusable environment
A reusable environment is created when you establish an assembler program as the main program by using either the old COBOL interfaces for preinitialization (RTEREUS runtime option), or the Language Environment interface, CEEPIPI.

routine
A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In Language Environment, refers to either a procedure, function, or subroutine.

*** routine-name**
A user-defined word that identifies a procedure written in a language other than COBOL.

*** run time**
The time at which an object program is executed. Synonymous with *object time*.

runtime environment
The environment in which a COBOL program executes.

*** run unit**
A stand-alone object program, or several object programs, that interact by means of COBOL `CALL` or `INVOKE` statements and function at run time as an entity.

A run unit is also called an enclave in Language Environment terminology.

S

SBCS
See *single-byte character set (SBCS)*.

scope terminator
A COBOL reserved word that marks the end of certain `PROCEDURE DIVISION` statements.It can be either explicit (`END-ADD`, for example) or implicit (separator period).

*** section**
A set of zero, one, or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**
A combination of words followed by a separator period that indicates the beginning of a section in any of these divisions: `ENVIRONMENT`, `DATA`, or `PROCEDURE`. In the `ENVIRONMENT DIVISION` and `DATA DIVISION`, a section header is composed of reserved words followed by a separator period. The permissible section headers in the `ENVIRONMENT DIVISION` are:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The permissible section headers in the `DATA DIVISION` are:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
```

In the `PROCEDURE DIVISION`, a section header is composed of a section-name, followed by the reserved word `SECTION`, followed by a separator period.

*** section-name**
A user-defined word that names a section in the PROCEDURE DIVISION.

segmentation
A feature of Enterprise COBOL that is based on the 85 COBOL Standard segmentation module. The segmentation feature uses priority-numbers in section headers to assign sections to fixed segments or independent segments. Segment classification affects whether procedures contained in a segment receive control in initial state or last-used state.

selection structure
A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

*** sentence**
A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**
A program that, together with its contained programs, is compiled separately from all other programs.

*** separator**
A character or two or more contiguous characters used to delimit character strings.

*** separator comma**
A comma (,) followed by a space used to delimit character strings.

*** separator period**
A period (.) followed by a space used to delimit character strings.

*** separator semicolon**
A semicolon (;) followed by a space used to delimit character strings.

sequence structure
A program processing logic in which a series of statements is executed in sequential order.

*** sequential access**
An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**
A file with sequential organization.

*** sequential organization**
The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search
A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

session bean
In EJB, an enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. (Oracle)

77-level-description-entry
A data description entry that describes a noncontiguous data item that has level-number 77.

*** sign condition**
The proposition (for which a truth value can be determined) that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

signature
(1) The name of an operation and its parameters. (2) The name of a method and the number and types of its formal parameters.

*** simple condition**
Any single condition chosen from this set:

- Relation condition
- Class condition

- Condition-name condition
- Switch-status condition
- Sign condition

See also *condition* and *negated simple condition*.

single-byte character set (SBCS)
A set of characters in which each character is represented by a single byte. See also *ASCII* and *EBCDIC (Extended Binary-Coded Decimal Interchange Code)*.

slack bytes (within records)
Bytes inserted by the compiler between data items to ensure correct alignment of some elementary data items. Slack bytes contain no meaningful data. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment.

slack bytes (between records)
Bytes inserted by the programmer between blocked logical records of a file, to ensure correct alignment of some elementary data items. In some cases, slack bytes between records improve performance for records processed in a buffer.

*** sort file**
A collection of records to be sorted by a format 1 SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**
An entry in the FILE SECTION of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

*** SOURCE-COMPUTER**
The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, where the source program is compiled, is described.

*** source computer entry**
An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION; this entry contains clauses that describe the computer environment in which the source program is to be compiled.

*** source item**
An identifier designated by a SOURCE clause that provides the value of a printable item.

source program
Although a source program can be represented by other forms and symbols, in this document the term always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement and terminates with the end program marker, if specified, or with the absence of additional source program lines.

source unit
A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as a *compilation unit*.

special character
A character that belongs to the following set:

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
$	Currency sign
,	Comma
;	Semicolon

Character	Meaning
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe
(	Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

SPECIAL-NAMES
The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

*** special names entry**
An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; this entry provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

*** special registers**
Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

*** standard data format**
The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

*** statement**
A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

structured programming
A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

*** subclass**
A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheritor or inheriting class; the superclass is the inheritee or inherited class.

*** subject of entry**
An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

*** subprogram**
See *called program*.

*** subscript**
An occurrence number that is represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**
An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

substitution character
A character that is used in a conversion from a source code page to a target code page to represent a character that is not defined in the target code page.

*** superclass**
A class that is inherited by another class. See also *subclass*.

surrogate pair
In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition
The proposition (for which a truth value can be determined) that an UPSI switch, capable of being set to an on or off status, has been set to a specific status.

*** symbolic-character**
A user-defined word that specifies a user-defined figurative constant.

syntax
(1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

*** system-name**
A COBOL word that is used to communicate with the operating environment.

T

*** table**
A set of logically consecutive items of data that are defined in the `DATA DIVISION` by means of the `OCCURS` clause.

*** table element**
A data item that belongs to the set of repeated items comprising a table.

text deck
Synonym for *object deck* or *object module*.

*** text-name**
A user-defined word that identifies library text.

*** text word**
A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or pseudo-text that is any of the following characters:

- A separator, except for space; a pseudo-text delimiter; and the opening and closing delimiters for alphanumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of alphanumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word `COPY` bounded by separators that are neither a separator nor a literal.

thread
A stream of computer instructions (initiated by an application within a process) that is in control of a process.

token
In the COBOL editor, a unit of meaning in a program. A token can contain data, a language keyword, an identifier, or other part of the language syntax.

top-down design
The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

top-down development
See *structured programming*.

trailer-label
(1) A data-set label that follows the data records on a unit of recording medium. (2) Synonym for *end-of-file label*.

troubleshoot
To detect, locate, and eliminate problems in using computer software.

*** truth value**
The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

typed object reference
A data-name that can refer only to an object of a specified class or any of its subclasses.

U

*** unary operator**
A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

unbounded table
A table with `OCCURS` *`integer-1`* `to UNBOUNDED` instead of specifying *integer-2* as the upper bound.

Unicode
A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. There are multiple encoding schemes to represent Unicode, including UTF-8, UTF-16, and UTF-32. Enterprise COBOL supports Unicode using UTF-16 in big-endian format as the representation for the national data type.

Uniform Resource Identifier (URI)
A sequence of characters that uniquely names a resource; in Enterprise COBOL, the identifier of a namespace. URI syntax is defined by the document *Uniform Resource Identifier (URI): Generic Syntax*.

unit
A module of direct access, the dimensions of which are determined by IBM.

universal object reference
A data-name that can refer to an object of any class.

unrestricted storage
Storage below the 2 GB bar. It can be above or below the 16 MB line. If it is above the 16 MB line, it is addressable only in 31-bit mode.

*** unsuccessful execution**
The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but can affect status indicators.

UPSI switch
A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

URI
See *Uniform Resource Identifier (URI)*.

*** user-defined word**
A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

*** variable**
A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

variable-length item
A group item that contains a table described with the DEPENDING phrase of the OCCURS clause.

*** variable-length record**
A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable-occurrence data item**
A variable-occurrence data item is a table element that is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry or be subordinate to such an item.

*** variably located group**
A group item following, and not subordinate to, a variable-length table in the same record. The group item can be an alphanumeric group or a national group.

*** variably located item**
A data item following, and not subordinate to, a variable-length table in the same record.

*** verb**
A word that expresses an action to be taken by a COBOL compiler or object program.

volume
A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

volume switch procedures
System-specific procedures that are executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM file system
A file system that supports COBOL sequential, relative, and indexed organizations.

W

web service
A modular application that performs specific tasks and is accessible through open protocols like HTTP and SOAP.

white space
Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

*** word**
A character string of not more than 30 characters that forms a user-defined word, a system-name, a reserved word, or a function-name.

*** WORKING-STORAGE SECTION**
The section of the DATA DIVISION that describes WORKING-STORAGE data items, composed either of noncontiguous items or WORKING-STORAGE records or of both.

workstation
A generic term for computers, including personal computers, 3270 terminals, intelligent workstations, and UNIX terminals. Often a workstation is connected to a mainframe or to a network.

wrapper
An object that provides an interface between object-oriented code and procedure-oriented code. Using wrappers lets programs be reused and accessed by other systems.

X

x
The symbol in a PICTURE clause that can hold any character in the character set of the computer.

XML
Extensible Markup Language. A standard metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to write applications to handle document types, author and manage structured information, and transmit and share structured information across diverse computing systems. The use of XML does not require the robust applications and processing that is necessary for SGML. XML is developed under the auspices of the World Wide Web Consortium (W3C).

XML data
Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

XML declaration
XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

XML document
A data object that is well formed as defined by the W3C XML specification.

XML namespace
A mechanism, defined by the W3C XML Namespace specifications, that limits the scope of a collection of element names and attribute names. A uniquely chosen XML namespace ensures the unique identity of an element name or attribute name across multiple XML documents or multiple contexts within an XML document.

XML schema
A mechanism, defined by the W3C, for describing and constraining the structure and content of XML documents. An XML schema, which is itself expressed in XML, effectively defines a class of XML documents of a given type, for example, purchase orders.

Z

z/OS UNIX file system
A collection of files and directories that are organized in a hierarchical structure and can be accessed by using z/OS UNIX.

zoned decimal data item
An external decimal data item that is described implicitly or explicitly as USAGE DISPLAY and that contains a valid combination of PICTURE symbols 9, S, P, and V. The content of a zoned decimal data item is represented in characters 0 through 9, optionally with a sign. If the PICTURE string specifies a sign and the SIGN IS SEPARATE clause is specified, the sign is represented as characters + or -. If SIGN IS SEPARATE is not specified, the sign is one hexadecimal digit that overlays the first 4 bits of the sign position (leading or trailing).

#

85 COBOL Standard
The COBOL language defined by the following standards:

- *ANSI INCITS 23-1985, Programming languages - COBOL*, as amended by *ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL* and *ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL*
- *ISO 1989:1985, Programming languages - COBOL*, as amended by *ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module* and *ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL*

2002 COBOL Standard
The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL*

2014 COBOL Standard
The COBOL language defined by the following standard:

- *INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

List of resources

Enterprise COBOL for z/OS

COBOL for z/OS publications

You can find the following publications in the Enterprise COBOL for z/OS library:

- *What's new*
- *Customization Guide*, SC27-8712-01
- *Language Reference*, SC27-8713-01
- *Programming Guide*, SC27-8714-01
- *Migration Guide*, GC27-8715-01
- *Performance Tuning Guide*, SC27-9202-00
- *Messages and Codes*, SC27-4648-00
- *Program Directory*, GI13-4526-01
- *Licensed Program Specifications*, GI13-4532-01

Softcopy publications

The following collection kits contain Enterprise COBOL and other product publications. You can find them at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss.

- *z/OS Software Products Collection*
- *z/OS and Software Products DVD Collection*

Support

If you have a problem using Enterprise COBOL for z/OS, see the following site that provides up-to-date support information: https://www.ibm.com/support/home/product/B984385H82239E03/Enterprise_COBOL_for_z/OS.

Related publications

z/OS library publications

You can find the following publications in the z/OS library.

Run-Time Library Extensions

- *Common Debug Architecture Library Reference*
- *Common Debug Architecture User's Guide*
- *DWARF/ELF Extensions Library Reference*

z/Architecture

- *Principles of Operation*

z/OS DFSMS

- *Access Method Services for Catalogs*
- *Checkpoint/Restart*

- *Macro Instructions for Data Sets*
- *Using Data Sets*
- *Utilities*

z/OS DFSORT

- *Application Programming Guide*
- *Installation and Customization*

z/OS ISPF

- *Dialog Developer's Guide and Reference*
- *User's Guide Vol I*
- *User's Guide Vol II*

z/OS Language Environment

- *Concepts Guide*
- *Customization*
- *Debugging Guide*
- *Language Environment Vendor Interfaces*
- *Programming Guide*
- *Programming Reference*
- *Run-Time Messages*
- *Run-Time Application Migration Guide*
- *Writing Interlanguage Communication Applications*

z/OS MVS

- *JCL Reference*
- *JCL User's Guide*
- *Programming: Callable Services for High-Level Languages*
- *Program Management: User's Guide and Reference*
- *System Commands*
- *z/OS Unicode Services User's Guide and Reference*
- *z/OS XML System Services User's Guide and Reference*

z/OS TSO/E

- *Command Reference*
- *Primer*
- *User's Guide*

z/OS UNIX System Services

- *Command Reference*
- *Programming: Assembler Callable Services Reference*
- *User's Guide*

z/OS XL C/C++

- *Programming Guide*
- *Run-Time Library Reference*

CICS Transaction Server for z/OS

You can find the following publications in the CICS library:

- *Developing CICS Applications*
- *API (EXEC CICS) Reference*
- *Developing CICS System Programs*
- *Global User Exit Reference*
- *XPI Reference*
- *Using EXCI with CICS*

COBOL Report Writer Precompiler

- *Programmer's Manual*, SC26-4301
- *Installation and Operation*, SC26-4302

Db2 for z/OS

You can find the following publications in the Db2 library:

- *Application Programming and SQL Guide*
- *Command Reference*
- *SQL Reference*

IBM Debug for z/OS (formerly IBM Debug for z Systems and IBM Debug Tool for z/OS)

You can find information about IBM Debug for z/OS in the IBM Debug for z/OS library.

IBM Debug for z/OS supersedes IBM Debug for z Systems® and IBM Debug Tool for z/OS. Not all references to IBM Debug for z Systems and IBM Debug Tool for z/OS have been changed in the COBOL documentation library. It is recommended that you upgrade your debugger to the latest level in order to have the full range of debugging features available. In some cases, you must upgrade your debugger to a certain version depending on what level of Enterprise COBOL you are using to create the COBOL application:

- IBM Debug Tool V13.1 supports Enterprise COBOL V5.1 and earlier versions
- IBM Debug for z Systems V14.0 supports Enterprise COBOL V6.1 and earlier versions
- IBM Debug for z Systems V14.1 supports Enterprise COBOL V6.2 and earlier versions

To find out which IBM debug product best suits your needs, see https://www.ibm.com/support/knowledgecenter/SSQ2R2_14.2.0/com.ibm.debug.cg.doc/common/dcompo.html?sc=SSQ2R2_latest.

IBM Developer for z/OS (formerly IBM Developer for z Systems)

You can find information about IBM Developer for z Systems in the IBM Developer for z/OS library.

Note: IBM Developer for z/OS supersedes IBM Developer for z Systems and Rational® Developer for z Systems.

You can find the following publications by searching their publication numbers in the IBM Publications Center.

IMS

- *Application Programming API Reference*, SC18-9699
- *Application Programming Guide*, SC18-9698

WebSphere® Application Server for z/OS

- *Applications*, SA22-7959

Softcopy publications for z/OS

The following collection kit contains z/OS and related product publications:

- *z/OS CD Collection Kit*, SK3T-4269

Java

- *IBM SDK for Java - Tools Documentation*, publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp
- *The Java 2 Enterprise Edition Developer's Guide*, download.oracle.com/javaee/1.2.1/devguide/html/DevGuideTOC.html
- *Java 2 on z/OS*, www.ibm.com/servers/eserver/zseries/software/java/
- *The Java EE 5 Tutorial*, download.oracle.com/javaee/5/tutorial/doc/
- *The Java Language Specification, Third Edition*, by Gosling et al., java.sun.com/docs/books/jls/
- *The Java Native Interface*, download.oracle.com/javase/1.5.0/docs/guide/jni/
- *JDK 5.0 Documentation*, download.oracle.com/javase/1.5.0/docs/

JSON

- JavaScript Object Notation (JSON), www.json.org

Unicode and character representation

- *Unicode*, www.unicode.org/
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

XML

- *Extensible Markup Language (XML)*, www.w3.org/XML/
- *Namespaces in XML 1.0*, www.w3.org/TR/xml-names/
- *Namespaces in XML 1.1*, www.w3.org/TR/xml-names11/
- *XML specification*, www.w3.org/TR/xml/

Product Number: 5655-EC6

SC27-9202-00