COBOL

Millennium Language Extensions Guide

GC26-9266-04

COBOL

Millennium Language Extensions Guide

GC26-9266-04

Note!

Before using this information and the products it supports, be sure to read the general information under “Notices” on page vii.

Fifth Edition (January 1999)

This edition applies to:

- IBM COBOL for OS/390 & VM (5648-A25) Version 2 Release 1 Modification 1 with APAR PQ20140 installed, and with VisualAge COBOL Millennium Language Extensions for OS/390 & VM (5648-MLE) Version 1 Release 1
- IBM COBOL Set for AIX (5765-548) Version 1 Release 1 Modification 1
- IBM VisualAge COBOL (5639-B92) Version 2.2 with FixPak 2
- IBM COBOL for MVS & VM (5688-197) Version 1 Release 2 Modification 2 with APAR PQ20139 installed, and with VisualAge COBOL Millennium Language Extensions for MVS & VM (5654-MLE) Version 1 Release 1

and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, HHX/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of these programs who wish to have information about them for the purpose of enabling: (1) the exchange of information between independently created programs and other programs (including these ones) and (2) the mutual use of the information which has been exchanged, should contact:

IBM Corporation, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

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

Programming Interface Information

This *IBM COBOL Millennium Language Extensions Guide* documents intended Programming Interfaces that allow the customer to write programs to obtain services of IBM COBOL millennium language extensions.

Notices

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
CICS
DB2
DFSORT
IBM
IMS
MVS
OS/2
OS/390
SQL/DS
System/390
VisualAge
VSE/ESA

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

About This Book

This book provides information on the millennium language extensions that have been incorporated into IBM COBOL to assist with Year 2000 processing.

The Year 2000 challenge (the Y2K problem or the Millennium Bug) results from applications using two digits rather than four to represent the year in date fields, and assuming that these values represent years from 1900 to 1999. This compact date format works well for the 1900s, but it does not work for the year 2000 and beyond because these applications interpret “00” as 1900 rather than 2000, producing incorrect results.

The millennium language extensions provide a compiler-assisted solution for COBOL programs. With these extensions, you can make minimal changes to programs that contain 2-digit year date fields, and recompile those programs to take advantage of compiler-assisted automatic date processing.

This book describes the new COBOL language elements introduced with the millennium language extensions, and shows how you can use them to help with date-related processing problems.

How This Book Will Help You

This book shows how to identify those applications that are candidates for a date windowing or date expansion solution, and gives guidance on how to implement such a solution.

The book assumes experience in developing and maintaining application programs, and knowledge of COBOL. It focuses on the COBOL constructs introduced with the millennium language extensions rather than on COBOL itself.

For information on the COBOL language, and the full syntax of all COBOL language elements, see the *IBM COBOL Language Reference* for your platform.

For information on how to use COBOL to develop applications, and how to compile and run COBOL programs, see the *IBM COBOL Programming Guide* for your platform.

Syntax Notation

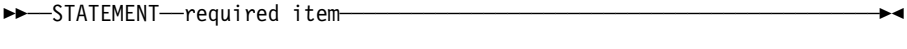
Throughout this book, syntax for COBOL statements, compiler options, and the Debug Tool is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

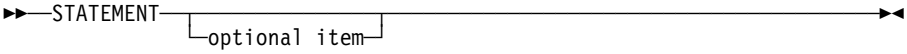
Symbol	Indicates
▸▸—	The syntax diagram starts here
—▸	The syntax diagram is continued on the next line
▸—	The syntax diagram is continued from the previous line
—▸◂	The syntax diagram ends here

Diagrams of syntactical units other than complete statements start with the ▸— symbol and end with the —▸ symbol.

- Required items appear on the horizontal line (the main path).

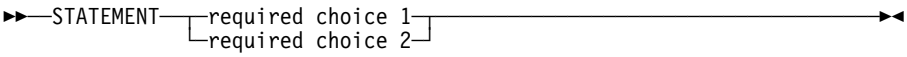


- Optional items appear below the main path.

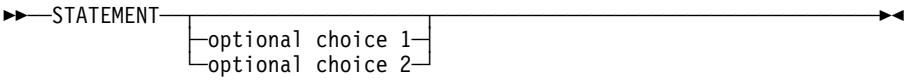


- When you can choose from two or more items, they appear vertically in a stack.

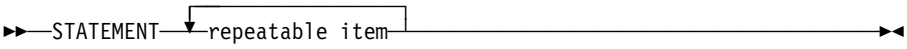
  If you **must** choose one of the items, one item of the stack appears on the main path.



  If choosing one of the items is optional, the entire stack appears below the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in uppercase letters (for example, PRINT). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, `item`). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, they must be entered as part of the syntax.
- Use at least one blank or comma to separate parameters.

Summary of Changes

This section lists the changes that have been made in this book since the first edition.

Changes in the Fourth Edition

This edition includes descriptions of:

- Enhancements to the millennium language extensions support:
  - Additional patterns for the DATE FORMAT clause, including *year-last* dates and YYX
  - DATE FORMAT for binary numeric items
  - Relaxation of the USING/RETURNING parameter rules for windowed dates
  - Special semantics for *trigger* and *limit* date values (host only).
- For the host platforms, new suboption TRIG/NOTRIG of the DATEPROC compiler option to enable or disable trigger and limit processing

Changes in the Third Edition (Softcopy only)

This edition includes descriptions of:

- Automated solutions for the OS/390 and MVS host and the Windows NT workstation.
- Conversion support available on Windows NT as a tool for updating COBOL source programs for year 2000 readiness.
- MLE compiler messages.
- CCCA DATE FORMAT clause support for VM.

Changes in the Second Edition

This edition includes descriptions of:

- COBOL for VSE/ESA (COBOL/VSE) as one of the supported COBOL compilers.
- COBOL and CICS Command Level Conversion Aid (CCCA) as a tool for updating COBOL source programs for year 2000 readiness.
- Date-related COBOL intrinsic functions.
- Changes to the millennium language extensions:
  - Signed numeric date fields are now permitted
  - Alphanumeric MOVE of an expanded date field to a windowed date field is now permitted
  - A windowed date field can now be defined as a subordinate data item to an expanded date field

Chapter 1. Introduction to Millennium Language Extensions

What is this year 2000 challenge? Most programs and databases use two digits rather than four to represent the year in date fields. The century is assumed to be 19 and is not explicitly represented. This compact date format works well for the 1900s because most programs operate on the same assumption; but it doesn't work for the year 2000 and beyond, because programs typically interpret 00 as 1900 rather than 2000 and proceed to give incorrect calculations. For example, years after 2000, such as 01, 02, and 03, are interpreted to be numerically smaller than, and therefore preceding, years before 2000, such as 97, 98, and 99. Some programs might even interpret 00 as some value other than a year, such as “delete immediately,” “never delete,” or “invalid.”

Before you can fix your 2-digit year date data, you must select which approach to use for each application, and decide whether the millennium language extensions are applicable.

When is MLE a Year 2000 Solution?

MLE can help you to implement two approaches for handling date data:

- Inferring the correct century through a century window
- Expanding year fields to hold century information explicitly

The pros and cons of each of these approaches can be found in Chapter 2, “Using the Millennium Language Extensions” on page 11. For a solution involving either of these approaches, or any combination of the two, the millennium language extensions offered with the IBM COBOL compilers may significantly reduce your source-code updates, and thereby improve your productivity.

Inferring the Correct Century Using a Century Window

With this approach, you establish a 100-year interval, called a century window, within which you assume all 2-digit years lie. Programs can then infer the correct century, depending on where in the century window the 2-digit year lies. For example, if the range of your century window is 1956 through 2055, and the 2-digit year is greater than or equal to 56, the century is inferred to be 19; otherwise, if the 2-digit year is less than or equal to 55, the century is inferred to be 20.

There are two types of century window: the fixed window and the sliding window. The fixed window doesn't change after it has been established; the sliding window moves forward with the system date.

Expanding Date Fields

With this approach, you copy the contents of a 2-digit year date field to another, larger, 4-digit year date field. The millennium language extensions can help you by automatically filling in the century part of the year in the receiving field, based on the century window.

You can use this technique for a number of approaches, but we'll concentrate on two in this book:

- Write a one-time COBOL program to expand your files and databases so that all the date fields have 4-digit years. For a simple example of a program that does this, see “File Conversion” on page 102. After expanding the files and databases, modify the programs that use them to specify corresponding data declarations with expanded-format dates.
- Change your existing programs to expand the date fields from the input records into Working-Storage fields, and use those work fields to process the dates in the main body of the programs. Then copy them back to the output records before writing them out. This technique is called *internal bridging*, and is described in “Internal Bridging” on page 15.

What is MLE?

MLE is short for Millennium Language Extensions, IBM's patent-pending technology that provides support for automated date windowing in its year 2000 ready COBOL and PL/I compilers. These extensions give you a mechanism to indicate to the compiler which dates should be windowed. It is a compiler-assisted solution for your windowed dates.

Using Language Environment on MVS, OS/390, VM, or VSE gives you windowing support at run time by supplying callable service routines to set and query a century window. But MLE provides even more assistance, essentially providing the logic that you would otherwise have to code yourself.

With MLE, you have the ideal situation: the ability to fix programs with few, if any, Procedure Division changes. While century windowing is the quickest and most powerful feature, field expansion is also supported, making MLE the easiest way to implement permanent internal bridges.

Compilers that Support MLE

Before starting to change any of your programs to use MLE, you need to ensure that you have the correct level of software to support the changes. IBM provides support for MLE in the following compilers:

- IBM COBOL for OS/390 & VM, with VisualAge COBOL Millennium Language Extensions for OS/390 & VM
- IBM COBOL Set for AIX
- IBM VisualAge COBOL
- IBM COBOL for MVS & VM, with VisualAge COBOL Millennium Language Extensions for MVS & VM
- IBM COBOL for VSE/ESA, with VisualAge COBOL Millennium Language Extensions for VSE/ESA

For a full list of the associated product numbers, releases, and modifications, see the Edition Notice on the inside front cover of this book. If you are unsure whether your compiler supports MLE, you should consult your software support group.

What You Gain from MLE

Using MLE, you can change your application data definitions in the Data Division to indicate which data items represent windowed or expanded dates, and, for windowed dates, how you want the window to be defined. The compiler can then automatically implement the associated windowing logic, in many cases without your having to code any changes in the Procedure Division. Because it is implemented directly by compiler-generated machine code, it is the best-performing windowing solution. So, where windowing is your approach of choice, MLE can help reduce the programming time and effort needed to implement a windowing solution and simplify subsequent maintenance of your application programs.

MLE also enables a flexible and comprehensive mixing of windowed dates with expanded dates within a program, allowing you to:

- Pick and choose which approach is best for each date data item within the program
- Use the windowing feature to quickly extend existing programs to run through the year 2000
- Use the expanded date field feature for long-term migration and the eventual expansion of dates in databases and files

You can use MLE to help with date field identification by identifying obvious date data items and letting the compiler diagnostics lead to you the ones that you missed.

Using MLE may also reduce your testing effort, particularly in those cases where you have few if any manual logic code changes. It's rather like using pre-tested logic.

Limitations of MLE

MLE is **not** a silver bullet. It is important to understand that, whatever the merits of MLE, it does not render unnecessary all the year 2000 work at your enterprise. A windowing solution might not be an acceptable approach for all your applications. While it can relieve you of many of the logic changes where you do choose windowing, it does not relieve you of the assessment, planning, analysis, implementation, and testing activities. There might be cases where using MLE requires you to make manual logic changes.

Also, the extensions are not intended to be a durable piece of language design, with a long-term usage. They do not constitute a fully-specified set of date-oriented data types, with rich semantics, enabling improved functionality for new applications or enhancements to existing applications. Their use should be limited to making year 2000 repairs only.

In particular, the windowing feature is not intended for long-term use. The start of the window must be in the range 1900–1999, so this feature buys you time to get code through the year 2000 and to a long-term solution that can be implemented later.

Because the extensions do not provide date data types, the non-year part of the supported date formats is denoted by Xs: no special processing is done for the non-year part. To do otherwise might change the meaning of existing programs. The only date-sensitive semantics that are provided involve automatically expanding (and contracting) the 2-digit year part of dates with respect to the 100-year window for the program.

For the COBOL contexts in which you cannot specify a windowed date, see “Impact on COBOL Language Elements” on page 87.

Where MLE is Applicable

Any COBOL application that is compiled with one of the MLE-supporting compilers can use MLE (for a list of compilers that support MLE, see “Compilers that Support MLE” on page 2). But consider these factors:

- The status of the programs: they should be stable programs that you are repairing for 2-digit years only
- What language level is being used (MLE requires COBOL 85 Standard level)
- What subsystems and files are involved

Using MLE for Year 2000 Work Only

The millennium language extensions were designed to meet a number of objectives in resolving date processing problems. To use the extensions effectively in your environment, you should evaluate the objectives that you need to meet, and compare them against the objectives of the millennium language extensions, to determine how your application can benefit from them.

The objectives of the millennium language extensions are as follows:

1. The primary objective is to extend the useful life of your application programs, as they are currently specified, into the twenty-first century.
2. Source changes to accomplish this must be held to the bare minimum, preferably limited to augmenting the declarations of date fields in the Data Division. To implement basic remediation of date problems, you should not be required to make any changes to the program logic in the Procedure Division.
3. The existing semantics of the programs will not be changed by the addition of date fields. For example, where a date is expressed as a literal, as in:

   ```
   If Date-of-Birth Less Than 450101 ...
   ```

   the programmer intended the literal 450101 to mean January 1, 1945, not 2045. The extensions retain this assumption about the program's original meaning, even though a *windowed* year value of 45 would be interpreted as the year 2045 (see “The Assumed Window” on page 32).

Source Conversion

The level of source code is an important consideration. If your programs have been compiled with the OS/VS COBOL compiler, you will have to convert the source, which is at the COBOL 68/74 Standard level, to COBOL 85 Standard. COBOL and CICS Command Level Conversion Aid (CCCA) can be a big help with the conversion, very often doing the entire job for you. Or, if you are going to use VisualAge COBOL for your year 2000 work, then its built-in facilities make source conversion easy.

If your programs have been compiled with VS COBOL II Release 3 or later, with the NOCMPR2 compiler option, no conversion is necessary. You can mix MLE programs with unchanged older modules, and only those programs with date logic need to be recompiled.

Note, however, that MLE is not available in the VS COBOL II compiler. Programs that use MLE must be compiled with one of the compilers that support MLE (see “Compilers that Support MLE” on page 2).

Subsystems with No Windowing

COBOL programs with MLE windowing can be used with any of the subsystems that normally support your applications, such as DB2, CICS, and VSAM. You can even specify windowed date fields as keys for sorting and merging if your sort software supports date windowing. However, some subsystem-specific functions do not support date windowing, and you cannot simply use windowed date fields in these contexts. Some examples of functions that don't support windowing are:

- VSAM file primary or alternate keys
- Search fields in database systems such as IMS or DB2
- Sort fields in DB2 (ORDER BY ...)
- Key fields in CICS commands

This can cause a variety of problems, from outright failure to confusion and errors from mismatched ordering. It is recommended to use expanded (4-digit year) date fields in these cases.

For information on using windowed date fields as sort keys for sorting and merging operations, see Chapter 5, “Using MLE with DFSORT” on page 44. For information on some of the advanced functions of MLE that can help circumvent these subsystem restrictions, see “Controlling Date Processing Explicitly” on page 25.

Try Out MLE

A pilot project is always something to consider for your year 2000 work. You should plan to do some preliminary testing with MLE to determine its usefulness in your environment. Migration to a year 2000 ready compiler is required if you are not already using one and are interested in taking the windowing approach.

Introduction to Millennium Language Extensions

MLE Scenario

If the century window approach can be used as all or part of the solution, then MLE is applicable. If MLE is applicable:

- Choose a fixed-window or sliding-window solution.
- Choose a century-window start year or offset for each program.
- Modify the data definition for each field by adding the appropriate DATE FORMAT clauses to the data definitions.

  If implementing the MLE solution on OS/390, MVS, or Windows NT, integrated tool suites are available to automate this for you. For details, see "What's the Easiest Way to Implement MLE?."
- Compile the program with the appropriate options and resolve any ambiguous references to the data items whose definitions were modified in the previous step. These ambiguous references will be highlighted by diagnostic messages.

  For a complete description of the diagnostic messages, see Appendix A, "MLE Messages" on page 110.

This process can be applied on a source module by source module basis. Any executable can be compiled and tested with full year 2000 support by using the compiler option to enable the new attributes. To test an application, use the following steps:

- Run regression tests with 1900 as the start year (the default); this should give the same results as those prior to the changes.
- Run date simulator tests with your selected start year.
- After each executable is tested, it can be moved into production with the year 2000 support enabled.

What's the Easiest Way to Implement MLE?

If using MLE is your approach of choice, IBM provides a full range of remediation tools to automate the process of identifying potential date fields and adding the DATE FORMAT clause to those date fields. On OS/390 and MVS/ESA, we provide the Millennium Language Extensions Automated Suite for OS/390 & MVS/ESA (MLE Automated Suite) and on Windows NT, the VisualAge COBOL Millennium Language Extensions Wizard (MLE Wizard).

It's important to note that, although these automated tool suites will simplify the source code updates, you must ensure the converted source compiles correctly and the results are as expected. This might require more than one iteration through the MLE process, particularly the date identification step. With MLE, you must identify ALL dates in your programs, not just the ones that are used in at-risk comparisons.

MLE Automated Suite

The MLE Automated Suite integrates IBM's Maintenance 2000 (MA2000), COBOL and CICS Command Level Conversion Aid (CCCA), and the Millennium Language Extensions to provide an efficient method for finding date-related fields in your source code

and automating the implementation of the compiler-assisted MLE technology. Figure 1 on page 7 illustrates how you use the Automated Suite to generate a data identification file and date remediated source.

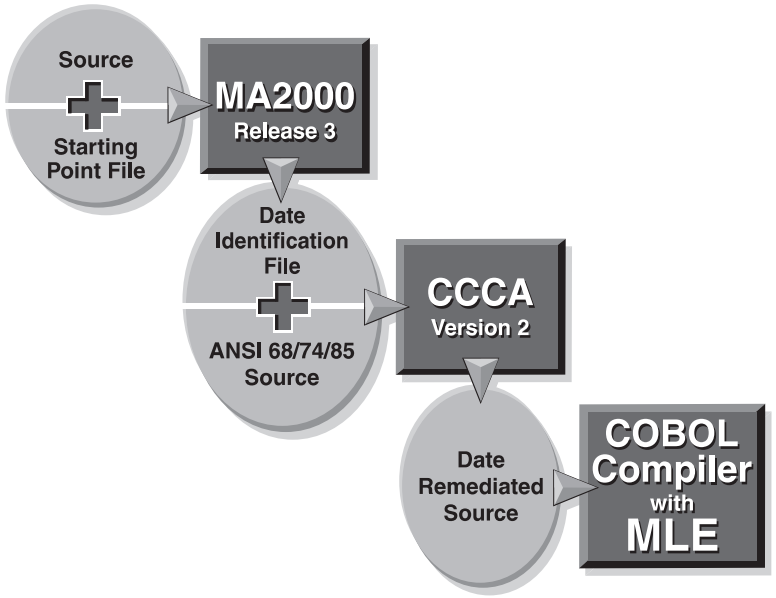


Figure 1. The MLE Automated Suite

First, you create a starting point file (also referred to as a conditions file), in which you define field names and attribute patterns for matching against the data items in the program. The conditions file and your source program are the inputs to MA2000. MA2000 generates the date identification file.

You then feed the date identification file and your source program into CCCA. Your source can be at any language level supported by CCCA. CCCA can add DATE FORMAT clauses, convert the source from 68/74 Standard to COBOL 85 Standard, or both.

Finally, compile your converted, date-remediated source using the MLE-enabled COBOL compiler. At this point, you need to tell the compiler to enable the windowing process (using the DATEPROC compiler option) and indicate the century window year (using the YEARWINDOW compiler option).

As with any other MLE coding changes, you'll need to review any diagnostic messages after the compilation step, resolve any ambiguous references, and recompile the program.

VisualAge COBOL Wizard

Included with VisualAge COBOL Version 2.2 FixPak 1 on the Windows NT operating system is the Millennium Language Extensions Wizard. The MLE Wizard is a fully-integrated solution for implementing the MLE technology using the VisualAge COBOL Year 2000 Analysis Tool, Program Conversion component, and MLE-enabled COBOL compiler. Figure 2 illustrates how the wizard generates date-remediated source code.

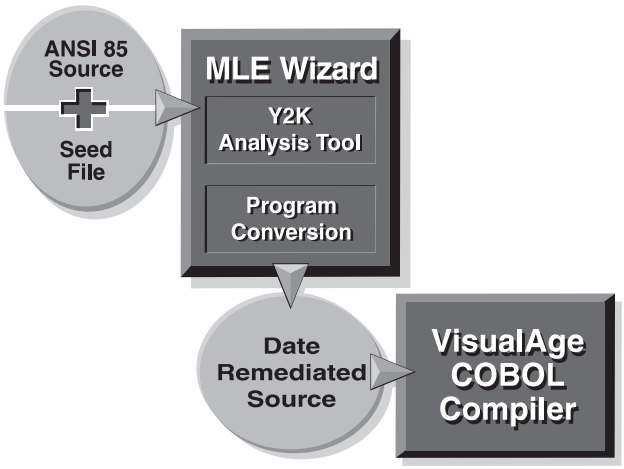


Figure 2. The VisualAge COBOL MLE Wizard

First, you create a seed file in which you define field names and attribute patterns for matching against the data items in the program. For more details, see "The Seed File" on page 35. The seed file and your COBOL 85 Standard level source programs are the inputs to MLE Wizard. (The MLE Wizard requires 85 Standard source. If your code is not at 85 Standard, you can run it through Program Conversion to upgrade it to COBOL 85 Standard level source prior to invoking the wizard.)

The Wizard will:

- Compile your programs to generate the SYSADATA file.
- Analyze your programs to find date fields.
- Convert your programs to add the DATE FORMAT clause to the date fields identified during the analysis phase.

You invoke the MLE Wizard from either:

- The Start menu. Select **Start - Programs - VisualAge COBOL for Windows - Millennium Language Extensions Wizard**.
- A COBOL project as a file-scoped action. Select the COBOL program(s) you want to convert, right mouse click, and choose the Millennium conversion action.

After the MLE Wizard completes successfully, it creates a project that contains the converted COBOL programs. You then compile the date-remediated source using the

VisualAge COBOL compiler. You'll need to tell the compiler to enable the windowing process (using the DATEPROC compiler option) and indicate the century window year (using the YEARWINDOW compiler option).

As with any other MLE process, you'll need to review any diagnostic messages after the compilation step, resolve any ambiguous references, and recompile the program.

But My Source isn't on OS/390, MVS, or Windows NT

If you're on a platform other than OS/390, MVS, or Windows NT, you can still take advantage of the automated tools available. It's even possible to mix and match tools and platforms to make the MLE changes as effortless as possible. For example, if your source code resides on VM and you have VisualAge COBOL on OS/2 and CCCA on VM, you can compile your source on VM to generate the SYSADATA file, download the file to OS/2 and use the Year 2000 Analysis Tool to create the date identification file (DIF), upload the date identification file back to the host to run through CCCA to add the DATE FORMAT clauses, and finally compile on VM with the MLE-enabled compiler.

Table 1 lists the tools available by platform.

Table 1 (Page 1 of 2). Tools to Help Implement the Millennium Language Extensions

Plat-form	Date Field Analysis	MLE Conversion	MLE Compile
Host			
OS/390	MA2000 (5655-A33)	CCCA (5648-B05)	MLE Product[1] (5648-MLE or 5654-MLE)
MVS	MA2000 (5655-A33)	CCCA (5648-B05)	MLE Product[1] (5654-MLE)
VSE		CCCA (5686-A07)	MLE Product[1] (5686-MLE)
VM		CCCA (5648-B05)	MLE Product[1] (5648-MLE or 5654-MLE)
Workstation			
AIX			Included in compiler
OS/2	Year 2000 Analysis component	[2]	Included in compiler

Table 1 (Page 2 of 2). Tools to Help Implement the Millennium Language Extensions

Plat-form	Date Field Analysis	MLE Conversion	MLE Compile
Windows	Year 2000 Analysis component	Program Conversion component	Included in compiler

Note:

[1] To compile with MLE, you need the COBOL compiler with PTFs plus the associated VisualAge COBOL Millennium Language Extensions product. For example, if you have the OS/390 Release 1 operating system and COBOL for MVS & VM (5648-197), you'll need PTFs for APAR PQ14261 plus VisualAge COBOL Millennium Language Extensions for MVS & VM (5654-MLE).

[2] Although Program Conversion is a component of VisualAge COBOL on both Windows and OS/2, DATE FORMAT conversion is available only on Windows NT.

Chapter 2. Using the Millennium Language Extensions

This chapter provides information on how you can use the millennium language extensions. It first shows a sample MLE scenario. It then outlines three approaches to implementing a year 2000 solution, and shows how COBOL date recognition facilities can help in each of these approaches. It also describes the coding changes you need to make in your COBOL programs to take advantage of the extensions.

Getting Started with MLE

Consider this short example which presents a problem and uses MLE in the solution.

A Simple Date Problem

Our example consists of a program which compares the date that a video tape was returned with the date it was due back to see whether to impose a fine. The two dates are stored in the file as 6-digit Gregorian dates; that is, YYMMDD.

```
01  Loan-Record.
    05  Member-Number   Pic X(8).
    05  Tape-ID         Pic X(8).
    05  Date-Due-Back   Pic X(6).
    05  Date-Returned   Pic X(6).
    ⋮
    If Date-Returned Greater than Date-Due-Back Then
       Perform Fine-Member.
```

If the tape was due back on September 14, 1998, the contents of Date-Due-Back are 980914. If it was returned on September 12, the contents of Date-Returned are 980912. Therefore, no fine is imposed, because Date-Returned is less than Date-Due-Back.

If we go forward in time, we can see how this reliable program behaves in January 2000. If the tape is due back on January 2, 2000, but is returned on December 31, 1999, the contents of the fields are:

Date-Due-Back 000102

Date-Returned 991231

In this case, Date-Returned is much larger than Date-Due-Back, so the program imposes a hefty fine for being 100 years late.

A Simple Solution

If the program recognized that the year 00 was in fact 2000, not 1900, the result would be different. In the new scenario, the “If ... Then ...” statement would behave properly, viewing 991231 as 19991231 and 000102 as 20000102, resulting in correct logic path and no fine for the tape that was returned early, not late.

IBM COBOL with MLE lets you do this quite easily, but you have to do a few things to make it work:

1. Decide on a century window, and tell the compiler about it. Take into account how far back your historical data goes and how far into the future your date data will be.

   If you decide to use 1950–2049 as your window, then COBOL evaluates the 2-digit year parts of your dates like this:

   Values 00 through 49 represent years 2000 through 2049.
   Values 50 through 99 represent years 1950 through 1999.

2. Tell the compiler which fields to treat as date fields, so it knows when to apply the century window and when not to. You do this by changing the COBOL copybook or source program, adding a DATE FORMAT clause to the date fields. Then the record layout looks like this:

   ```
   01  Loan-Record.
       05  Member-Number   Pic X(8).
       05  Tape-ID         Pic X(8).
       05  Date-Due-Back   Pic X(6) Date Format is yyxxxx.
       05  Date-Returned   Pic X(6) Date Format is yyxxxx.
   ```

   The two Ys in the date pattern yyxxxx means that the date field is a *windowed date field*, to which COBOL will apply the century window that was defined by the YEARWINDOW compiler option.

3. Use the two new compiler options to tell the compiler about the century window:

   DATEPROC Tells the compiler to enable the windowing process. For our example, we'll use DATEPROC(FLAG) to get all the date-related diagnostic messages.

   YEARWINDOW Tells the compiler what the century window is. For our example, we'll use YEARWINDOW(1950) to give us a fixed window of 1950–2049.

   To use a sliding window, specify a negative integer from -1 through -99. For example, YEARWINDOW(-48) defines a sliding window that starts 48 years before the year that the program is running. So, if the program is running in 1998, the century window is 1950-2049, and in 1999 it automatically becomes 1951-2000, and so on.

4. Compile the program. Check the diagnostic messages to verify that the statements will still work correctly.
5. Test the program.
6. Optionally, recompile the program with DATEPROC(NOFLAG) when you put the new version of the program into production. This gives you a clean compile listing for your records.

To summarize, with MLE it's quite straightforward to introduce a windowed date concept into a COBOL program by deciding on a century window, changing some data descriptions, and recompiling the program. It's even easier when using one of the auto-

| mated solutions, see “What's the Easiest Way to Implement MLE?” on page 6 for
| details.

Note: For field expansion, you will need to use MOVE or COMPUTE statements to copy the contents of windowed date fields to expanded date fields.

Large-Scale Use of MLE

As you can see from the previous discussion, it's a simple matter to change one COBOL program to take advantage of automatic date windowing with MLE. But your installation's programs will be more complicated than this simple example, and you will need to implement changes to many programs at a time. This section outlines the steps you should follow in converting a complete application, or a suite of related programs, to MLE.

Note that these steps are a guideline only; you will need to consider many other factors in formulating a conversion plan of this magnitude. Some other factors to consider are:

- Your installation's change control system and methodology
- The requirement to impose a “freeze” on other changes while this process is taking place
- The availability of testing environments
- The level of interaction between applications

With these and other factors in mind, you can formulate a conversion plan based on the following steps:

1. Choose an application or group of applications to be remediated using MLE.
2. Identify the century window to be used for the application. Take into account how far back your historical data goes and how far into the future your date data will be.
3. Identify the date fields that need to be expanded. For example, if a VSAM key is a date field, this field should be expanded rather than windowed. For information on expanding date fields, see “Full Field Expansion” on page 17.
4. Document which copybooks are used by which source programs.
5. Locate data items that contain dates in all of the copybooks and programs. This date identification should be accurate in that it should identify as many of the dates as possible, without identifying non-date fields as dates (false positives). This step should also try to identify the date patterns of these date data items.
6. Edit the copybooks and add DATE FORMAT clauses to the data definitions. You may have to look at the usage of the dates in COBOL programs to determine the date pattern to use.

   Note: You must identify all dates with DATE FORMAT clauses, even those not used in comparisons.
7. Compile the programs that have the most date usage first, to allow the compiler to help complete the date field identification for the copybooks. Use the DATEPROC(FLAG) compiler option and resolve any compiler diagnostics that are

produced. Most diagnostics will be resolved by adding more DATE FORMAT clauses.

8. Once the date fields are all marked with DATE FORMAT clauses in the copybooks, compile all of the programs in the application that use those copybooks. If more date fields are identified in the copybooks, then change these newly-identified fields and go back and recompile any programs that use those copybooks.

   Note: You must identify all dates with DATE FORMAT clauses, even those not used in comparisons.

9. Keep track of other applications that share these copybooks. These other applications may be compiled with the DATEPROC option at a later date, or, if they have been remediated with another method such as manual windowing, they may need to have NODATEPROC added to their CBL/PROCESS statements to avoid the need to re-analyze them.

Resolving Date-Related Logic Problems

This section discusses three approaches that you can adopt to assist with date-related processing problems, and shows how you can use the millennium language extensions with each approach to achieve a solution.

The approaches outlined here are:

- Basic remediation (the century window solution)
- Internal bridging
- Full field expansion

Basic Remediation

The simplest method of ensuring that your programs will continue to function through the year 2000 is to implement a century window solution.

With this method, you define a century window, and specify the fields that contain windowed dates. The compiler then interprets the 2-digit years in those date fields according to the century window.

In “A Simple Solution” on page 11, there is a sample program that shows how you can implement automatic date windowing capabilities by using the DATE FORMAT clause. The “after” version of the program is reproduced here with the changes highlighted:

```
CBL  DATEPROC(FLAG),YEARWINDOW(-60)
        ⋮
     01  Loan-Record.
         05  Member-Number   Pic X(8).
         05  Tape-ID         Pic X(8).
         05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
         05  Date-Returned   Pic X(6) Date Format yyxxxx.
         ⋮
         If Date-Returned > Date-Due-Back Then
            Perform Fine-Member.
```

In this program, there are no changes to the Procedure Division from the previous version. The addition of the DATE FORMAT clause on the two date fields means that the compiler recognizes them as windowed date fields, and therefore applies the century window when processing the IF statement. For example, if Date-Due-Back contains “000102” (January 2, 2000) and Date-Returned contains “991231” (December 31, 1999), Date-Returned is less than (earlier than) Date-Due-Back, so the program does not perform the Fine-Member paragraph.

A solution such as this is easy to implement, and may be the most appropriate solution for you if you have a large number of programs and very little time.

Advantages:

- Fast and easy to implement.
- No change to the program's logic, therefore less testing required.
- This solution will allow your programs to function into and beyond the year 2000.

Disadvantages:

- This should be regarded as a short-term solution, not as a permanent fix. (All programs will probably need to use expanded dates some day.)
- There may be some performance degradation introduced by the date windowing functions.
- Implementation of this solution is application-dependent. It might not suit all applications.

Internal Bridging

If your files and databases have not yet been converted to 4-digit year dates, but you prefer to use 4-digit expanded-year logic in your programs, you can use an internal bridge technique to process the dates. Your program might be structured as follows:

1. Read the input files with 2-digit year dates.
2. Declare these 2-digit dates as windowed date fields and move them to expanded date fields, so that the compiler automatically expands them to 4-digit year dates.
3. In the main body of the program, use the 4-digit year dates for all date processing.
4. Window the dates back to 2-digit years.
5. Write the 2-digit year dates to the output files.

This process provides a convenient migration path to a full expanded-date solution, and also may have performance advantages over using windowed dates.

Using this technique, your changes to the program logic are minimal. You simply add statements to expand and contract the dates, and change those statements that refer to dates to use the 4-digit year date fields in Working-Storage instead of the 2-digit year fields in the records.

Because you are converting the dates back to 2-digit years for output, you should allow for the possibility of the year being outside the century window. For example, if a date field contains the year 2020, but the century window is 1920–2019, then the date is outside the window, and simply moving it to a 2-digit year would be incorrect. To protect against this, you can use a COMPUTE statement to store the date, with the ON SIZE ERROR phrase to detect whether or not the date is within the century window. For more details, see “ON SIZE ERROR Phrase” on page 79.

The following example shows how a program can be changed to implement an internal bridge method:

```
CBL  DATEPROC(FLAG),YEARWINDOW(-60)
       ⋮
    File Section.
    FD  Customer-File.
    01  Cust-Record.
        05  Cust-Number     Pic 9(9) Binary.
        ⋮
        05  Cust-Date       Pic 9(6) Date Format yyxxxx.
    Working-Storage Section.
    77  Exp-Cust-Date       Pic 9(8) Date Format yyyyxxxx.
        ⋮
    Procedure Division.
        Open I-O Customer-File.
        Read Customer-File.
        Move Cust-Date to Exp-Cust-Date.
        ⋮
   *=====================================================*
   * Use expanded date in the rest of the program logic  *
   *=====================================================*
        ⋮
        Compute Cust-Date = Exp-Cust-Date
                On Size Error Display "Exp-Cust-Date outside
                                      century window"
                End-Compute
        Rewrite Cust-Record.
```

Advantages:

- Straightforward changes to the program logic, therefore testing is easy.
- This solution will allow your programs to function into and beyond the year 2000.
- This is a good incremental step towards a full expanded-year solution.
- Good performance.

Disadvantages:

- Some risk of data corruption.

Full Field Expansion

The full field expansion solution involves explicitly expanding 2-digit year date fields to contain full 4-digit years in your files and databases, and then using those fields in expanded form in your programs. This is the only method by which you can be assured of reliable date processing for all applications.

The millennium language extensions allow you to progressively move towards a full date field expansion solution, using the following steps:

1. Apply the short-term (basic remediation) solution, and use this until you have the resources to implement a more permanent solution.
2. Apply the internal bridging scheme. This allows you to use expanded dates in your programs while your files continue to hold dates in 2-digit year form. This in turn will allow you to progress more easily to a full field expansion solution, because there will be no further changes to the logic in the main body of the programs.
3. Change the file layouts and database definitions to use 4-digit year dates.
4. Change your COBOL copybooks to reflect these 4-digit year date fields.
5. Run a utility program (or special-purpose COBOL program) to copy from the old format files to the new format. For a sample program, see Figure 7 on page 102.
6. Recompile your programs and perform regression testing and date testing.

After you have completed the first two steps, the remaining steps in the sequence can be repeated any number of times. You do not need to change every date field in every file at the same time. Using this method, you can select files for progressive conversion based on criteria such as business needs or interfaces with other applications.

Advantages:

- This is a permanent solution; no more changes are required. This solution will allow your programs to function into and beyond the year 2000.
- Best performance.
- Maintenance will be easier.

Disadvantages:

- Need to ensure that changes to databases, copybooks, and programs are all synchronized.

Programming Techniques

This section describes the techniques you can use in your programs to take advantage of date processing, and the effects of using date fields on COBOL language elements.

For a more detailed description of COBOL language elements, see Chapter 7, “Millennium Language Extensions Reference” on page 66. For full details of COBOL syntax and restrictions, see the *IBM COBOL Language Reference* for your platform.

Date Comparisons

The millennium language extensions support year-last, year-first, and year-only date fields for the most common operations on date fields: comparisons, moving and storing, incrementing and decrementing.

Year-First and Year-Only Date Fields

A *year-first* date field is a date field whose DATE FORMAT specification consists of YY or YYYY, followed by one or more Xs. The date format of a *year-only* date field has just the YY or YYYY. When you compare two date fields of either of these types, the two dates must be compatible; that is, they must have the same number of non-year characters (see "Compatible Dates" on page 30). The number of digits for the year component need not be the same. For example:

```
77  Todays-Date         Pic X(8) Date Format yyyyxxxx.
01  Loan-Record.
    05  Date-Due-Back   Pic X(6) Date Format yyxxxx.
    ⋮
    If Date-Due-Back Greater than Todays-Date Then...
```

In this example, a windowed date field is compared to an expanded date field, so the century window is applied to Date-Due-Back.

Note that Todays-Date must have a DATE FORMAT clause in this case to define it as an expanded date field. If it did not, it would be treated as a non-date field, and would therefore be considered to have the same number of year digits as Date-Due-Back. The compiler would apply the assumed century window to it, which would create an inconsistent comparison. For more information, see "The Assumed Window" on page 32.

Level 88 Condition-Name

If a windowed date field has an 88-level condition-name associated with it, the literal in the VALUE clause is windowed against the century window for the compile unit rather than the assumed century window of 1900–1999. For example:

```
05  Date-Due        Pic 9(6) Date Format yyxxxx.
88  Date-Target              Value 051220.
```

If the century window is 1950–2049 and the contents of Date-Due is 051220 (representing December 20, 2005), then the condition

```
If Date-Target
```

would evaluate to TRUE, but the condition

```
If Date-Due = 051220
```

would evaluate to FALSE. This is because the literal 051220 is treated as a non-date, and therefore windowed against the assumed century window of 1900–1999 to represent December 20, 1905. But where the same literal is specified in the VALUE clause of an 88-level condition-name, it becomes part of the data item to which it is attached. Because this data item is a windowed date field, the century window is applied whenever it is referenced.

You can also use the DATEVAL intrinsic function in a comparison expression to convert a literal to a date field, and the output from the intrinsic function will then be treated as either a windowed or expanded date field to ensure a consistent comparison. For example, using the above definitions, both of these conditions

```
If Date-Due = Function DATEVAL (051220 "YYXXXX")
If Date-Due = Function DATEVAL (20051220 "YYYYXXXX")
```

would evaluate to TRUE. For more information on the DATEVAL intrinsic function, see "DATEVAL" on page 25.

Restriction: With a level-88 condition name, you can also specify the THRU option on the VALUE clause, for example:

```
05  Year-Field      Pic 99 Date Format yy.
88  In-Range               Value 98 Thru 06.
```

With this form, the windowed value of the second item must be greater than the windowed value of the first item. However, the compiler can only verify this if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-60)).

For this reason, if the YEARWINDOW compiler option specifies a sliding century window, you cannot use the THRU option on the VALUE clause of a level-88 condition name.

Note: The windowed order requirement does not apply to year-last date fields. If you specify a condition-name VALUE clause with the THROUGH phrase for a year-last date field, the two literals must follow normal COBOL rules. That is, the first literal must be less than the second literal.

Year-Last Date Fields

A year-last date field is a date field whose DATE FORMAT clause specifies one or more Xs *preceding* the YY or YYYY. Such date formats are commonly used to display dates, but are less useful computationally, because the year, which is the most significant part of the date, is in the least significant position of the date representation.

The main intent of providing year-last date formats is to let you identify and track such dates. However, year-last dates are also supported in a limited number of COBOL operations, and are fully supported as windowed keys in SORT or MERGE statements, assuming that your version of DFSORT, or equivalent, has the appropriate capabilities. For further details, see "Sorting and Merging" on page 23.

Apart from sort and merge operations, functional support for year-last date fields is limited to equal or unequal comparisons and certain kinds of assignment. The operands must either be dates with identical (year-last) date formats, or a date and a non-date. The compiler itself does not provide automatic windowing for any operation on year-last dates. Unsupported usage, such as arithmetic on year-last dates, is diagnosed with an error-level message.

If you need more general date processing capability for year-last dates, you should isolate and operate on the year part of the date, as outlined under “Other Date Formats” on page 24.

Triggers, Limits and Special Indicators (Host Only)

Triggers and limits are special values that never match valid dates because either the value is non-numeric or the non-year part of the value cannot occur in an actual date. Triggers and limits are recognized in date fields and also in non-dates used in combination with date fields.

For alphanumeric windowed date or year fields, the special values are HIGH-VALUE, LOW-VALUE, and SPACE. For alphanumeric and numeric windowed date fields with at least one X in the DATE FORMAT clause (that is, date fields other than just a year), values of all zeros or all nines are also treated as triggers or limits. The difference between a trigger and a limit is not in the particular value, but in the way it is used. You can use any of the special values as either a trigger or a limit.

When used as triggers, special values can indicate a specific condition such as “date not initialized” or “account past due.” When used as limits, special values are intended to act as dates earlier or later than any valid date. LOW-VALUE, SPACE and zeros are lower limits; HIGH-VALUE and nines are upper limits.

Trigger and limit support is activated by specifying the TRIG suboption of the DATEPROC compiler option. For more information, see “DATEPROC” on page 66. If the DATEPROC(TRIG) compiler option is in effect, automatic expansion of windowed date fields, prior to their use as operands in comparisons, arithmetic and so on, is sensitive to these special values.

Note: The DATEPROC(TRIG) option will result in slower performing code for windowed date comparisons. The DATEPROC(NOTRIG) option is a performance option that assumes valid date values in all windowed date fields.

When an actual or assumed windowed date field contains a trigger, it is expanded as if the trigger value were propagated to the century part of the expanded date result, rather than inferring 19 or 20 as the century value as in normal windowing. In this way, your application can test for special values or use them as upper or lower date limits. Specifying DATEPROC(TRIG) also enables SORT and MERGE statement support of the DFSORT special indicators, which correspond with triggers and limits.

As an example of using limits, suppose that your application checks subscriptions for expiration, but you want some subscriptions to last indefinitely. You can use the expiration date field to hold either normal expiration dates or a high limit for the “everlasting” subscription. For example, consider the following code fragment:

```
Process DateProc(Flag,Trig)...
⋮
  01 SubscriptionRecord.
⋮
     03 ExpirationDate Pic 9(6) Date Format yyxxxx.
⋮
  77 TodaysDate Pic 9(6) Date Format yyxxxx.
⋮
     If TodaysDate >= ExpirationDate
        Perform SubscriptionExpired
```

Suppose that today's date is January 4, 2000, represented in `TodaysDate` as 000104, and that one subscription record has a normal expiration date of December 31, 1999, represented as 991231, while the special subscription expiration date is coded as 999999. Because both dates are windowed, the first subscription is tested as if 20000104 were compared against 19991231, so the test succeeds. But, when the special value is detected, trigger expansion is used instead of windowing, so the test proceeds as if 20000104 were compared against 99999999, and thus fails, and will always fail.

Sign Condition

Some applications use special values such as zeros in date fields to act as a “trigger,” that is, to signify that some special processing is required. For example, in an Orders file, a value of zero in Order-Date might signify that the record is a customer totals record rather than an order record. The program compares the date to zero, as follows:

```
01  Order-Record.
    05  Order-Date      Pic S9(5) Comp-3 Date Format yyxxx.
    ⋮
    If Order-Date Equal Zero Then...
```

However, if you are compiling with the NOTRIG suboption of the DATEPROC compiler option, this comparison is not valid because the literal value Zero is a non-date, and is therefore windowed against the assumed century window to give a value of 1900000 (for more information, see “Treatment of Non-Dates” on page 31).

Alternatively, or if compiling on the workstation, you can use a sign condition as follows:

```
    If Order-Date Is Zero Then...
```

instead of a literal comparison. With a sign condition, Order-Date is treated as a non-date, and the century window is not considered.

Notes:

1. This only applies if the operand in the sign condition is a simple identifier rather than an arithmetic expression. If an expression is specified, the expression is evaluated first, with the century window being applied where appropriate. The sign condition is then compared to the results of the expression.
2. You could also use the UNDATE intrinsic function to achieve the same result. For details, see "UNDATE" on page 26.
3. For host programs, you could also use the TRIG suboption of the DATEPROC compiler option to achieve a similar result. For details, see "Triggers, Limits and Special Indicators (Host Only)" on page 20.

Date Arithmetic

You can perform arithmetic operations on date fields in the same manner as any numeric data item, and, where appropriate, the century window will be used in the calculation. However, there are some restrictions on where date fields can be used in arithmetic expressions.

The permissible arithmetic operations on date fields are:

- Adding a non-date to a date field to give a date field result. For example:

```
Add 1 to Ending-Year

Add 10 to Last-Review-Year Giving Next-Review-Year

Compute Start-School-Year = Year-of-Birth + 5
```

- Subtracting a non-date from a date field to give a date field result. For example:

```
Subtract 2 From Year-Of-Expiry

Subtract 1 From This-Year Giving Last-Year

Compute Year-Of-Birth = This-Year - Age
```

- Subtracting a date field from a compatible date field to give a non-date result. For example:

```
Subtract Year-of-Birth From This-Year Giving Age

Compute Years-To-Go = Expiry-Year - This-Year
```

For a discussion of arithmetic operations with date fields, and a list of the restrictions, see "Arithmetic Expressions" on page 77.

Sorting and Merging

Notes:

1. This section applies to host platforms only.
2. DFSORT is the IBM sort/merge licensed program. Wherever DFSORT is mentioned, any other equivalent sort/merge product can be used.

If your SORT product supports the Y2PAST option and the various windowed year field identifiers (Y2B, Y2C, Y2D, Y2S, and Y2Z), you can perform sort and merge operations using windowed date fields as sort keys. Virtually all date fields that can be specified with a DATE FORMAT clause are supported, including binary year fields and year-last date fields. The fields will be sorted in windowed year sequence, according to the century window specified by the YEARWINDOW compiler option.

If your SORT product also supports the date field identifiers Y2T, Y2U, Y2W, Y2X, and Y2Y then you can use the TRIG suboption of the DATEPROC compiler option. (Support for these date field identifiers was added to DFSORT/MVS with the application of APAR PQ19684.)

The special indicators that DFSORT recognizes exactly match the triggers and limits supported by COBOL, that is, LOW-VALUE, HIGH-VALUE, and SPACE for alphanumeric date or year fields, and all zeros and all nines for numeric and alphanumeric date fields with at least one non-year digit.

The following example shows a sort of a transaction file, with the transaction records being sorted by date and time within account number. The field Trans-Date is a windowed Julian date field.

```
SD  Transaction-File
    Record Contains 29 Characters
    Data Record is Transaction-Record.

01  Transaction-Record.
    05  Trans-Account   Pic 9(8).
    05  Trans-Type      Pic X.
    05  Trans-Date      Pic 9(5) Date Format yyxxx.
    05  Trans-Time      Pic 9(6).
    05  Trans-Amount    Pic 9(7)V99.
    ⋮
    Sort Transaction-File
         On Ascending Key Trans-Account
                          Trans-Date
                          Trans-Time
         Using Input-File
         Giving Sorted-File.
```

COBOL passes the relevant information to DFSORT in order for it to perform the sorting operation properly. In addition to the information that is always passed to DFSORT, COBOL also passes:

- The century window as the Y2PAST sort option

- The windowed year field and date format of Trans-Date

DFSORT then uses this information to perform the sorting process.

For more information and examples of COBOL sort coding, see Chapter 5, “Using MLE with DFSORT” on page 44. For information on DFSORT and the Y2PAST option, see the *DFSORT Application Programming Guide* for your platform.

Other Date Formats

To be eligible for automatic windowing, a date field should contain a 2-digit year as the first or only part of the field. The remainder of the field, if present, must be between 1 and 4 characters, but its content is not important. For example, it can contain a 3-digit Julian day, or a 2-character identifier of some event specific to the enterprise.

If there are date fields in your application that do not fit these criteria, then you may have to make some code changes to define just the year part of the date as a date field with the DATE FORMAT clause. Some examples of these types of date formats are:

- A 7-character field consisting of a 2-digit year, 3 characters containing an abbreviation of the month and 2 digits for the day of the month. This is not supported because date fields can have only 1 through 4 non-year characters.
- A Gregorian date of the form DDMMYY. Automatic windowing is not provided because the year component is not the first part of the date. Year-last dates such as these are fully supported as windowed keys in SORT or MERGE statements, and are also supported in a limited number of other COBOL operations.

If you need to use date windowing in cases like these, you will need to add some code to isolate the year portion of the date.

In the following example, the two date fields contain dates of the form DDMMYY:

```
03  Last-Review-Date Pic 9(6).
03  Next-Review-Date Pic 9(6).
⋮
Add 1 to Last-Review-Date Giving Next-Review-Date.
```

In this example, if Last-Review-Date contains 230197 (January 23, 1997), then Next-Review-Date will contain 230198 (January 23, 1998) after the ADD statement is executed. This is a simple method of setting the next date for an annual review. However, if Last-Review-Date contains 230199, then adding 1 gives 230200, which is not the desired result.

Because the year is not the first part of these date fields, the DATE FORMAT clause cannot be applied without some code to isolate the year component. In the next example, the year component of both date fields has been isolated so that COBOL can apply the century window and maintain consistent results:

```
03  Last-Review-Date Date Format xxxxyy.
    05  Last-R-DDMM  Pic 9(4).
    05  Last-R-YY    Pic 99 Date Format yy.
03  Next-Review-Date Date Format xxxxyy.
    05  Next-R-DDMM  Pic 9(4).
    05  Next-R-YY    Pic 99 Date Format yy.
⋮
Move Last-R-DDMM to Next-R-DDMM.
Add 1 to Last-R-YY Giving Next-R-YY.
```

Controlling Date Processing Explicitly

There may be times when you want COBOL data items to be treated as date fields only under certain conditions, or only in specific parts of the program. Or your application may contain 2-digit year date fields that cannot be declared as windowed date fields because of some interaction with another software product. For example, if a date field is used in a context where it is only recognized by its true binary contents without further interpretation, the date in this field cannot be windowed. This includes:

- A key on a VSAM file
- A search field in a database system such as IMS or DB2
- A key field in a CICS command

Conversely, there may be times when you want a date field to be treated as a non-date in specific parts of the program.

COBOL provides two intrinsic functions to cater for these conditions:

DATEVAL Converts a non-date into a date field

UNDATE Converts a date field into a non-date

DATEVAL

You can use the DATEVAL intrinsic function to convert a non-date into a date field, so that COBOL will apply the relevant date processing to the field. The first argument to the function is the non-date to be converted, and the second argument specifies the date format. The second argument is a literal string with a specification similar to that of the date pattern in the DATE FORMAT clause.

As an example, assume that a program contains a field Date-Copied, and that this field is referenced many times in the program, but most of these references simply move it between records or reformat it for printing. Only one reference relies on it containing a date, for comparison with another date.

In this case, it is better to leave the field as a non-date, and use the DATEVAL intrinsic function in the comparison statement. For example:

```
03  Date-Distributed Pic 9(6) Date Format yyxxxx.
03  Date-Copied      Pic 9(6).
⋮
If Function Dateval(Date-Copied "YYXXXX") Less than
                    Date-Distributed ...
```

In this example, the DATEVAL intrinsic function converts Date-Copied into a date field so that the comparison will be meaningful.

In most cases, the compiler makes the correct assumption about the interpretation of a non-date, but accompanies this assumption with a warning-level diagnostic message. This typically happens when a windowed date is compared to a literal:

```
03  When-Made        Pic x(6) Date Format yyxxxx.
⋮
If When-Made = "850701" Perform Warranty-Check.
```

The literal is assumed to be a compatible windowed date but with a century window of 1900–1999, thus representing July 15, 1985. You can use the DATEVAL intrinsic function to make the year of the literal date explicit, and eliminate the warning message:

```
If When-Made = Function Dateval("19850701" "YYYYXXXX")
    Perform Warranty-Check.
```

For a full description and syntax of the DATEVAL intrinsic function, see “DATEVAL Intrinsic Function” on page 81, and the *IBM COBOL Language Reference* for your platform.

UNDATE

The UNDATE intrinsic function converts a date field to a non-date, so that it can be referenced without any date processing.

Warning: The use of UNDATE should be avoided, except as a last resort, since the compiler could 'lose' the flow of date fields in your program. This could result in date comparisons not being windowed properly. Use more DATE FORMAT clauses instead of function UNDATE for MOVE and COMPUTE.

In the following example, the field Invoice-Date in Invoice-Record is a windowed Julian date. In some records, it contains a value of "00999" to indicate that this is not a “true” invoice record, but a record containing file control information.

Invoice-Date has been given a DATE FORMAT clause because most of its references in the program are date-specific. However, in the instance where it is checked for the existence of a control record, the value of "00" in the year component will lead to some confusion. A year of "00" in Invoice-Date will represent a “true” year of either 1900 or 2000, depending on the century window. This is compared to a non-date (the literal "00999" in the example), which will always be windowed against the assumed century window and will therefore always represent the year 1900.

To ensure a consistent comparison, you should use the UNDATE intrinsic function to convert Invoice-Date to a non-date. This means that the IF statement is not comparing any date fields, so it does not need to apply any windowing. For example:

```
01  Invoice-Record.
    03  Invoice-Date    Pic x(5) Date Format yyxxx.
    ⋮
    If FUNCTION UNDATE(Invoice-Date) Equal "00999" ...
```

For a full description and syntax of the UNDATE intrinsic function, see “UNDATE Intrinsic Function” on page 84, and the *IBM COBOL Language Reference*.

Analyzing Date-Related Diagnostic Messages

When the DATEPROC(FLAG) compiler option is in effect, the compiler produces diagnostic messages for every statement that defines or references a date field. As with all compiler-generated messages, each date-related message has one of the following severity levels:

- Information-level, to draw your attention to the definition or use of a date field.
- Warning-level, to indicate that the compiler has had to make an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.
- Error-level, to indicate that the usage of the date field is incorrect. Compilation continues, but run-time results are unpredictable.
- Severe-level, to indicate that the usage of the date field is incorrect. The statement that generated this error is discarded from the compilation.

You should always eliminate error-level and severe-level messages from your program by correcting the use of the date fields in the affected statements. Warning-level messages deserve special attention because the assumptions that the compiler makes may not be correct.

Your approach to warning-level diagnostic messages should be:

1. Avoidance. Be specific in your program changes, so that the compiler does not need to make assumptions about your intended use of date fields (see “Avoiding Warning-Level Messages”).
2. Analysis. Examine each diagnostic message, and either eliminate it, or, where you consider it to be unavoidable, ensure that the compiler's assumptions are correct (see “Analyzing Warning-Level Messages” on page 28).

For additional details on how to use the MLE messages, see Appendix A, “MLE Messages” on page 110.

Avoiding Warning-Level Messages

To avoid warning-level diagnostic messages, follow these simple guidelines:

- Add DATE FORMAT clauses to any data items that will contain date data, even if they are not used in comparisons.
- Don't specify a date field in a context where a date field doesn't make sense, such as a FILE STATUS, PASSWORD, ASSIGN USING, LABEL RECORD, or LINAGE item. If you do, you'll get a warning-level message and the date field will be treated as a non-date.
- Ensure that implicit or explicit aliases for date fields are compatible, such as in a group item that consists solely of a date field.

- Ensure that if a date field is defined with a VALUE clause, the value is compatible with the date field definition.
- Use the DATEVAL intrinsic function if you want a non-date treated as a date field, such as when moving a non-date to a date field, or comparing a windowed date field with a non-date and you want a windowed date comparison. If you don't use DATEVAL, the compiler will make an assumption about the use of the non-date, and produce a warning-level diagnostic message. Even if the assumption is correct, you may want to use DATEVAL just to eliminate the message. For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 25.

Analyzing Warning-Level Messages

The following guidelines will help you to analyze date-related warning-level diagnostic messages:

- The diagnostic messages may indicate some date data items that should have had a DATE FORMAT clause but were missed. You should either add DATE FORMAT clauses to these items, or use the DATEVAL intrinsic function in references to them.
- Pay particular attention to the usage of literals in relation conditions involving date fields or in arithmetic expressions that include date fields. Note that the DATEVAL function may be used on literals (as well as non-date data items) to specify a DATE FORMAT pattern to be used, and the UNDATE function may be used to enable a date field to be used in a context where date-oriented behavior is not desired.
- With the REDEFINES and RENAMES clauses, the compiler may produce a warning-level diagnostic message if a date field and a non-date occupy the same storage location. You should check these cases carefully to confirm that all uses of the various aliased data items are correct, and that none of the perceived non-date redefinitions actually are dates or can adversely affect the date logic in the program.

Other Potential Problems

When you change a COBOL program to make use of the millennium language extensions, you may find that some parts of the program need special attention to resolve unforeseen changes in behavior. This section outlines some of those areas that you may need to consider.

Packed Decimal Fields

COMPUTATIONAL-3 fields (packed decimal format) are often defined as having an odd number of digits, even if the field will not be used to hold a number of that magnitude. This is because the internal representation of packed decimal numbers always allows for an odd number of digits (for details of internal representation, see the *IBM COBOL Programming Guide* for your platform).

A field that holds a 6-digit Gregorian date, for example, may be declared as PIC S9(6) COMP-3, and this will reserve four bytes of storage. But the programmer may have

declared the field as PIC S9(7), knowing that this would reserve the same four bytes, with the high-order digit always containing a zero.

If you simply add a DATE FORMAT YYXXXX clause to this field, the compiler will give you a diagnostic message because the number of digits in the PICTURE clause does not match the size of the date format specification (see “PICTURE Clause” on page 90). In this case, you need to check carefully each use of the field. If the high-order digit is never used, you can simply change the field definition to PIC S9(6). If it is used (for example, if the same field can hold a value other than a date), you need to take some other action. Other actions you can take include:

- Using a REDEFINES clause to define the field as both a date and a non-date (this will also produce a warning-level diagnostic message)
- Defining another Working-Storage field to hold the date, and moving the numeric field to the new field
- Not adding a DATE FORMAT clause to the data item, and using the DATEVAL intrinsic function when referring to it as a date field

Contracting Moves

When you move an expanded alphanumeric date field to a windowed date field, the move does not follow the normal COBOL conventions for alphanumeric moves. When both the sending and receiving fields are date fields, the move is right-justified, not left-justified as normal. For an expanded-to-windowed (contracting) move, this means that the leading two digits of the year are truncated.

Depending on the contents of the sending field, the results of such a move may be incorrect. For example:

```
77  Year-Of-Birth-Exp    Pic x(4) Date Format yyyy.
77  Year-Of-Birth-Win    Pic xx   Date Format yy.
    ⋮
    Move Year-Of-Birth-Exp to Year-Of-Birth-Win.
```

If Year-Of-Birth-Exp contains "1925", Year-Of-Birth-Win will contain "25". However, if the century window is 1930–2029, subsequent references to Year-Of-Birth-Win will treat it as 2025, which is incorrect.

Concepts

This section describes some of the concepts of the millennium language extensions, and how they interact with other parts of COBOL. For an understanding of the principles on which these concepts are based, see “Using MLE for Year 2000 Work Only” on page 4.

Date Semantics

All arithmetic, whether performed on date fields or not, acts only on the numeric contents of the fields; date semantics for the non-year parts of date fields are not provided. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

However, date semantics *are* provided for the year parts of date fields. For example, if the century window is 1950–2049, and the value of windowed date field TwoDigitYear is 49, then the following ADD statement will result in the SIZE ERROR imperative statement being executed:

```
Add 1 to Two-Digit-Year
  on Size Error Perform Century-Window-Overflow
End-Add
```

Compatible Dates

The meaning of the term *compatible dates* depends on the COBOL division in which the usage occurs, as follows:

- The Data Division usage is concerned with the declaration of date fields, and the rules governing COBOL language elements such as subordinate data items and the REDEFINES clause. In the following example, Review-Date and Review-Year are compatible because Review-Year can be declared as a subordinate data item to Review-Date:

  ```
  01  Review-Record.
      03  Review-Date            Date Format yyxxxx.
          05  Review-Year Pic XX Date Format yy.
          05  Review-M-D  Pic XXXX.
  ```

  For full details, see the *IBM COBOL Language Reference*

- The Procedure Division usage is concerned with how date fields can be used together in operations such as comparisons, moves, and arithmetic expressions. For year-first and year-only fields only, to be considered compatible, date fields must have the same number of non-year characters. For example, a field with DATE FORMAT YYXXXX is compatible with another field that has the same DATE FORMAT, and with a YYYYXXXX field, but not with a YYXXX field.

  Year-last date fields must have identical date formats. In particular, operations between windowed date fields and expanded year-last date fields are not allowed. For example, you can move a date field with a date format of `XXXXYY` to another `XXXXYY` date field, but not to a date field with a format of `XXXXYYYY`.

The remainder of this discussion relates to the Procedure Division usage of compatible dates for year-first and year-only date formats.

You can perform operations on date fields, or on a combination of date fields and non-dates, provided that the date fields in the operation are compatible. For example, with the following definitions:

```
01  Date-Gregorian-Win  Pic 9(6) Packed Date Format yyxxxx.
01  Date-Julian-Win     Pic 9(5) Packed Date Format yyxxx.
01  Date-Gregorian-Exp  Pic 9(8) Packed Date Format yyyyxxxx.
```

The statement:

```
If Date-Gregorian-Win Less than Date-Julian-Win ...
```

is inconsistent because the number of non-year digits is different between the two fields. The statement:

```
If Date-Gregorian-Win Less than Date-Gregorian-Exp ...
```

is accepted because the number of non-year digits is the same for both fields. In this case the century window is applied to the windowed date field (Date-Gregorian-Win) to ensure that the comparison is meaningful.

Where a non-date is used in conjunction with a date field, the non-date is either assumed to be compatible with the date field, or treated as a simple numeric value, as described in the following section.

Treatment of Non-Dates

The simplest kind of non-date is just a literal value. The following items are also non-dates:

- A data item whose data description does not include a DATE FORMAT clause.
- The results (intermediate or final) of some arithmetic expressions. For example, the difference of two date fields is a non-date, whereas the sum of a date field and a non-date is a date field.
- The output from the UNDATE intrinsic function.

When you use a non-date in conjunction with a date field, the compiler interprets the non-date as either a date whose format is compatible with the date field, or a simple numeric value. This interpretation depends on the context in which the date field and non-date are used, as follows:

Comparison

Where a date field is compared to a non-date, the non-date is considered to be compatible with the date field in the number of year and non-year characters. In the following example:

```
01  Date-1              Pic 9(6) Date Format yyxxxx.
    ⋮
    If Date-1 Greater than 971231 ...
```

Because the non-date literal 971231 is being compared to a windowed date field, it is treated as if it had the same DATE FORMAT as Date-1, but with a base year of 1900.

Arithmetic operations

In all supported arithmetic operations, non-dates are treated as simple numeric values. In the following example:

```
01  Date-2              Pic 9(6) Date Format yyxxxx.
    ⋮
    Add 10000 to Date-2.
```

the numeric value 10000 is added to the Gregorian date in Date-2, effectively adding one year to the date.

MOVE statement

Moving a date field to a non-date is not supported. However, you can use the UNDATE intrinsic function to achieve this. For more information, see “UNDATE” on page 26.

When you move a non-date to a date field, the sending field is assumed to be compatible with the receiving field in the number of year and non-year characters. For example, when you move a non-date to a windowed date field, the non-date field is assumed to contain a compatible date with a 2-digit year.

The Assumed Window

Where the program operates on windowed date fields, the compiler applies the century window for the compile unit; that is, the one defined by the YEARWINDOW compiler option. Where a windowed date field is used in conjunction with a non-date, and the context demands that the non-date also be treated as a windowed date, the compiler uses an assumed century window to resolve the non-date.

The assumed century window is 1900–1999, which is typically not the same as the century window for the compile unit.

In many cases, particularly for literal non-dates, this assumed century window will be the correct choice. For example, in the construct:

```
01  Manufacturing-Record.
    03  Makers-Date Pic X(6) Date Format yyxxxx.
    ⋮
    If Makers-Date Greater than "720101" ...
```

the literal should retain its original meaning of January 1, 1972, and not change to 2072 if the century window is, for example, 1975–2074. Even if the assumption is correct, it is better to make the year explicit, and also eliminate the warning-level diagnostic message that accompanies application of the assumed century window, by using the DATEVAL intrinsic function:

```
    If makers-Date Greater than
                 Function Dateval("19720101" "YYYYXXXX") ...
```

In other cases however, the assumption may not be correct. For example:

```
01  Project-Controls.
    03  Date-Target     Pic 9(6).
    ⋮
01  Progress-Record.
    03  Date-Complete   Pic 9(6) Date Format yyxxxx.
    ⋮
    If Date-Complete Less than Date-Target ...
```

For this example, assume that Project-Controls is in a copybook that is used by other applications that have not yet been upgraded for year 2000 processing, and therefore Date-Target cannot have a DATE FORMAT clause. In the example, if:

- The century window is 1910–2009,

- Date-Complete is 991202 (Gregorian date: December 2, 1999), and
- Date-Target is 000115 (Gregorian date: January 15, 2000),

then:

- Date-Complete is earlier than (less than) Date-Target.

However, because Date-Target does not have a DATE FORMAT clause, it is a non-date, so the century window applied to it is the assumed century window of 1900–1999, which means that it is processed as January 15, 1900. So Date-Complete will be greater than Date-Target, which is not the desired result.

In this case, you should use the DATEVAL intrinsic function to convert Date-Target to a date field for this comparison. For example:

```
If Date-Complete Less than
   Function Dateval (Date-Target "YYXXXX") ...
```

For more information on the DATEVAL intrinsic function, see “DATEVAL” on page 25.

Chapter 3. Using the VisualAge COBOL Year 2000 Analysis Tool

This chapter gives a brief overview of the VisualAge COBOL Year 2000 Analysis Tool, and describes how you can use it to find date fields in your COBOL programs.

More information on using the Impact Analysis Tool can be found in the on-line help for VisualAge COBOL, and in the following publications:

- For OS/2, *VisualAge COBOL Getting Started on OS/2*
- For Windows, *VisualAge COBOL Getting Started on Windows*

Background

VisualAge COBOL Year 2000 Analysis Tool (Impact Analysis) scans COBOL programs looking for fields that are likely to contain dates, and reports on where these fields occur in the program. Impact Analysis gets its initial search criteria from a user-defined seed file, which contains generic field names and attributes of potential date fields.

Impact Analysis identifies potential date fields by:

- Matching field names and attributes in the seed file with data items in the program
- Detecting data items used as arguments to, or receiving values from, date-related intrinsic functions
- Detecting data items used to receive date values in ACCEPT ... FROM DATE or ACCEPT ... FROM DAY statements
- Propagating from a field identified by any of the above methods to another field, because of the context of its use

Although Impact Analysis is a PC-based product, it is not restricted to COBOL programs for the workstation only; programs for the host environment can also be scanned.

Impact Analysis runs in the following environments:

Windows 95
Windows NT
OS/2

It scans the SYSADATA output file from a compiled COBOL program to find date fields, and produces output files and reports showing the fields that are likely to contain dates. For host-based programs, the SYSADATA file (not the COBOL source file) is downloaded to the workstation for analysis.

Impact Analysis does not support programs containing object-oriented statements.

Impact Analysis produces output in the form of reports, browsable HTML files, and plain text files.

Using Impact Analysis

The following process outlines the steps involved in analyzing one program:

1. Select the program to be analyzed and compile it with the ANALYZE and ADATA compiler options. These options have the following effect on the compile process:

   ANALYZE The compiler looks for embedded SQL and CICS statements in the program, and checks their syntax.

   COPY/BASIS/REPLACE processing is performed, regardless of the setting of the LIB compiler option.

   No executable code is produced.

   ADATA The compiler produces a SYSADATA file, which contains information about the program. It is this SYSADATA file that is scanned by Impact Analysis to find date fields.

   For full information on these compiler options, see the *IBM COBOL Programming Guide* for your platform.

2. For host platforms, download the SYSADATA file to the workstation. The file on the workstation should have a file extension of `.ADT`, and it should be downloaded in binary format so that no ASCII translation is performed.

3. Create a seed file that contains information about fields in the program. A sample seed file (`sample.xsd`) is provided with Impact Analysis, and you should use this as a starting point for your own file. For an example of a seed file, see "The Seed File."

4. Run Impact Analysis. There are two ways to start the analysis process:

   - By entering the `y2krepor` command at an OS/2 or Windows command prompt. For a description of the `y2krepor` command and its parameters, see the Impact Analysis reference material.
   - From a graphical interface. You can start the graphical interface from the COBOL Tools folder on the workstation desktop, from the Project menu of WorkFrame, or by entering the `dczybat` command. You can then use the graphical interface to define the parameters for the run, and to start the analysis process.

5. Use the output from the run to further modify and refine the seed file, and repeat the Impact Analysis run.

6. The output from Impact Analysis is now available for review, and you can use it as a guide to changing your COBOL program to define date fields.

The Seed File

The seed file provides a mechanism for identifying date fields in a COBOL program. It contains one or more seeds, each of which defines field name and attribute patterns for matching against the data items in the program.

Following is an example of a seed in a seed file:

```
<seed>
  <pattern> *date*
  <except> *update*
  </pattern>
  <pattern> *today*
  <pattern> *yesterday*
  <pattern> *tomorrow*
  <attribute> XXXXXX
  <attribute> 999999
</seed>
```

The <pattern> tags define search criteria for matching against the names of fields in the program. In this example:

- Any field with the string “date” in its name is a possible date field, unless the string “date” is part of the larger string “update”
- Any field with the string “yesterday”, “today”, or “tomorrow” in its name is a possible date field

Note that the seed patterns, like COBOL data names, are not case-sensitive. Therefore *DATE*, *Date*, and *date* are all treated the same.

Any field whose name matches one of these criteria is then checked against the <attribute> tag. In this example, if the field is defined as any of:

```
Pic XXXXXX
Pic X(6)
Pic 999999
Pic 9(6)
```

the field is considered to be a date field.

Within the <seed> tag, you can indicate the parts of date fields that contain year and non-year data, using the <used-as-year> and <used-as-non-year> tags. These tags define the starting position and length of the year and non-year components of the date field. In the following example, all fields of length 6 with the string “DATE” in their name are assumed to contain Gregorian dates of the form YYMMDD.

```
<seed>
  <pattern> *DATE*
  <attribute> 999999
  <attribute> XXXXXX
  <used-as-year> <index> 1 <length> 2 </used-as-year>
  <used-as-non-year> <index> 3 <length> 4 </used-as-non-year>
</seed>
```

Analysis Output

The analysis run produces a number of output files:

- Plain text file (.XRL)
- Browsable HTML file (.HTM)

- Formatted reports (`.XRF` and `.XRT`) for printing or browsing
- Date identification file for input to CCCA (`.XCA`)

The formatted reports and the CCCA file are optional, and are only produced if the relevant options are specified either on the `y2krepor` command or the graphical interface.

Interpreting the Results

The Impact Analysis documentation describes the format and content of the output files, and you should refer to that for an understanding of the information. The information produced relates to many aspects of the scanned program, not just date fields. This section describes the date-related information.

Output Format

The output format discussed here will be that found in the browsable HTML file (`.HTM`).

The file contains a number of tags that identify the program and its components. The `<RESULT>` tag and its subordinate tags define the results for date-related data. The subordinate tags of `<RESULT>` that are of interest here are:

`<KIND>` This indicates whether a field is used as a date field in the program. It can contain:

`<USED-AS-YEAR>`
: The result may be year-related. This may have come directly from the seed file, or it may have been inferred from the context. An example of an inference of this form is:

  - The seed file defines `Date-Of-*` as a date field candidate
  - The field `Date-Of-Order` is flagged as a date field because of this seed
  - The program contains the statement:

    ```
    Move Date-Of-Order to When-Ordered.
    ```

  - The field `When-Ordered` is *inferred* to be a date field because of this usage

`<ALWAYS-YEAR>`
: The result is always year-related. This result can only come from the seed file; it cannot be inferred.

`<USED-AS-NON-YEAR>`
: The result is either directly seeded or inferred to be non-year related.

`<ALWAYS-NON-YEAR>`
: The result is non-year related, as defined in the seed file.

`<USED-AS-YEAR-AND-NON-YEAR>`
: The result is used in different ways in different parts of the program.

<YEAR-REASON>
States the reason that the field (or part of the field) is year-related.

<NON-YEAR-REASON>
States the reason that the field (or part of the field) is not year-related.

Using the information given in these tags, you should be able to modify your program to take advantage of the millennium language extensions. As a guide, your modifications should be:

- Add DATE FORMAT clauses to the <ALWAYS-YEAR> fields, provided that the fields are of a format supported by the DATE FORMAT clause (see "Other Date Formats" on page 24).
- Examine the <USED-AS-YEAR> fields to see whether they should also have DATE FORMAT clauses.
- Examine the <USED-AS-YEAR-AND-NON-YEAR> fields to see whether they should have DATE FORMAT clauses, or whether you should define new fields to use in some instances.

Remember that Impact Analysis is a tool to help you determine the year 2000 impact on your programs; its results can only be as good as the contents of the seed file. Whether you make these changes manually or use another automation tool to update your programs, you should always perform extensive testing to ensure that the changes achieve the desired effect, and do not cause any other unrelated or unforeseen problems.

Cross-Compilation Analysis

Cross-compilation analysis is used to analyze multiple, related program units together. The information gathered is saved in a DB2 database and is used across the analysis of all input .ADT files.

The input to Impact Analysis cross compilation will be a set of related ADATA files (*.ADT) and their input seed files (*.XSD). Your initial seed files can be modeled on the sample seed file (sample.xsd) shipped with the product.

You first build a plain text file (*.XIN) with each line of the file containing the name of one ADATA file (*.ADT) and its associated seed file (*.XSD). For example:

```
order01.adt  order01.xsd
order02.adt  order02.xsd
order03.adt  order03.xsd
invoic01.adt  invoic01.xsd
invoic02.adt  invoic02.xsd
```

Optionally, a single seed file (*.XSD) can be specified via the command line as the default for all input ADATA files (*.ADT) which do not have a specific seed file. If no seed file is specified, the filename of the seed file is assumed to be the same as the filename of the ADATA file.

Using the VisualAge COBOL Year 2000 Analysis Tool

To begin cross-compilation analysis, either enter the following command:

`y2kxcomp /`*`filename`*`.xin`

or use the graphical interface.

Analysis output will be produced in plain text files (`*.XRL`) and browsable HTML files (`*.HTM`). Formatted report files (`*.XRF` and `*.XPT`) may also be produced.

Chapter 4. MLE and CCCA

CCCA (COBOL and CICS Command Level Conversion Aid) is a tool that can assist with conversion from earlier levels of COBOL. Its primary function is to convert source programs from COBOL 68 or 74 Standard to COBOL 85 Standard. It reads COBOL source programs and copybooks and detects COBOL language constructs that are no longer supported in the current release of the compiler. It changes these language constructs and rewrites the source and copybooks to produce new, up-to-date COBOL source programs.

CCCA is available on the following platforms:

- OS/390, MVS, and VM (Program number 5648-B05)
- VSE/ESA (Program number 5686-A07)

Although the primary function of CCCA is to upgrade COBOL programs to the latest level, it can also add DATE FORMAT clauses to COBOL source code to provide a fast and easy start to making your programs year 2000 ready. These two functions can be performed together in a single process, or, if your COBOL source is already at the COBOL 85 Standard level, CCCA can add DATE FORMAT clauses as an independent process.

This chapter gives you a brief overview of the MLE functions of CCCA, and shows how you can use CCCA to help upgrade your programs to year 2000 readiness. For full information, see the *CCCA User's Guide* for your platform.

Note: For Windows NT, VisualAge COBOL's Conversion component provides function similar to CCCA with the ability to both upgrade source language level and add the DATE FORMAT clause to identified date fields. With VisualAge COBOL, you can use the Year 2000 Analysis Tool (see Chapter 3, “Using the VisualAge COBOL Year 2000 Analysis Tool” on page 34 for details) to generate a date identification file, which is used as input to Conversion.

VisualAge COBOL takes the MLE conversion process one step further with the Millennium Language Extensions Wizard. The MLE Wizard integrates the Year 2000 Analysis Tool and Conversion components to automate adding DATE FORMAT clauses to identified date fields. For details, see “VisualAge COBOL Wizard” on page 8.

The CCCA Conversion Process

The following outline shows how you can use CCCA to apply date formats to a COBOL program. This is a very simplified approach; for full details of CCCA and its capabilities, you should refer to the *CCCA User's Guide* for your platform.

1. Ensure that the program and copybooks are error-free. This is a requirement of both CCCA and MLE; if there are errors in the post-conversion program, they are easier to resolve if they can only have been introduced by the conversion process.
2. Create or obtain a date identification file that contains the names of COBOL data items to be given a DATE FORMAT clause (see “The Date Identification File” on

page 41). You can create this file (or library member) manually using an editor such as ISPF or ICCF, or you can use a tool such as VisualAge COBOL Year 2000 Analysis Tool to produce the file for you. For information on the Impact Analysis Tool, see Chapter 3, “Using the VisualAge COBOL Year 2000 Analysis Tool” on page 34.

3. Access CCCA. See your CCCA documentation for information on how to access CCCA on your platform.
4. Set CCCA environment options. This is part of the CCCA user interface, accessed through the CCCA on-line menus. You use it to define such things as (for OS/390 and MVS) the high-level qualifiers for your shared and private data sets, and (for VSE) the library member type of your date identification file.
5. Set language levels. This is a CCCA panel where you nominate the language levels of the source and target programs.
6. Set conversion options. There are two “Conversion Options” panels, which you use to specify such things as the level of diagnostics you want printed, and whether or not CCCA is to generate new sequence numbers on the target COBOL source program. Two of these options are of interest in the MLE environment:

   Check procedure names
   should be set to N (no) for improved performance.

   Add DATE FORMAT clause to date fields
   should be set to Y (yes) to indicate that this is a DATE FORMAT conversion run.
7. Using the “Conversion” panels (for OS/390 and MVS, there are three panels; for VSE/ESA, there is just one), specify the files or libraries that contain the COBOL source programs and copybooks to be converted, the name of the date identification file, and the files or libraries to contain the converted programs and copybooks.
8. Submit the conversion job.
9. Compile the converted source. This will be an MLE-enabled compilation, so you need to ensure that the DATEPROC and YEARWINDOW compiler options are in effect. CCCA performs this step automatically for you if you selected the **Compile after converting** option on the “Conversion Options” panel (step 6). In this case, the compiler options used are DATEPROC(FLAG) and the installation default value of the YEARWINDOW option.
10. Check the compilation listing for errors. If necessary, make other changes manually to the converted program.

The Date Identification File

The date identification file is a required input file for the CCCA DATE FORMAT conversion process. It contains the names of COBOL data items that are to be treated as date fields, and the date formats of those fields. The date identification file must be a fixed-length file with a record length of 80 bytes. The information in the file can be free-format, but fixed-format (data in columns) is recommended for readability.

An example of a date identification file is:

```
* CCCA conversion run for August 1998
* Invoice system
<ORDINV01>
  134 YYXXXX    Invoice-Date
  136 YYXXXX    Order-Date
  328 YYXXXX    Transaction-Date Of Account-Record
  412 YYXXXX    Transaction-Date Of Invoice-Record
  820 YYYYXXXX  Todays-Date
* Equipment service system
<SERVMT32>
   70 YYXXX     Service-Date Of Service-History Of Equipment-Record
   78 YYXXX     Delivery-Date Of Equipment-Record
```

The contents of each item in the file is:

- Comment lines, denoted by an asterisk in column 1.
- The program name, enclosed in less than and greater than signs. The information from here to the next program name (or the end of the file if this is the last program name in the file) will apply to this program only.
- The line number in the program of where the data item is defined. This does not have to be the actual line number in the program; CCCA simply requires a numeric field as a delimiter.
- A date format specification, as used in the DATE FORMAT clause. This must be in a form supported by the millennium language extensions.
- The name of the data item. This can be an elementary item or a group item, but it should be a field that contains a date in a format supported by the DATE FORMAT clause.
- If the data item cannot be uniquely identified without qualification, use the keyword OF (“IN” is not allowed) followed by the name of a data item at a higher level. This can be repeated as many times as needed to uniquely identify the field.

Applying Date Formats

When CCCA finds a data item in the COBOL source that is listed in the date identification file, it attempts to apply a DATE FORMAT clause to that data item. The date format given to the field is the format defined specifically for that field in the date identification file.

Before applying a DATE FORMAT clause to the data description entry for a data item, CCCA performs various checks on the syntax. For example, for an elementary item, it checks that the number of picture characters matches the length of the date format specification. If CCCA finds that an invalid syntax would result, it produces a diagnostic message stating the reason the DATE FORMAT clause was not added.

CCCA Limitations

You should not expect to be able to compile and run these modified COBOL programs without testing and, probably, further manual changes. CCCA and MLE are simply tools to help you with your year 2000 conversion process; they do not constitute an all-embracing solution.

Specifically, the CCCA MLE conversion process:

- Changes only the Data Division of the program, and only to add DATE FORMAT clauses where instructed. It does not change the definition of any data items in any other way.
- Does not take into account all of the COBOL language rules that define where a DATE FORMAT clause can and cannot appear, and the interactions between the DATE FORMAT clause and other Data Division clauses (see “Data Division Considerations” on page 68).
- Does not change anything in the Procedure Division. That is, it does not change the format or structure of any program statements, and it does not add any system intrinsic calls.

Because of this, you may find that when you compile a CCCA-converted program, the compilation listing contains diagnostic messages because of inconsistencies introduced with the date fields. You should always compile your programs after the CCCA conversion process, either using your normal compile procedures or the CCCA **Compile after converting** option, and manually make whatever changes are necessary to complete the program conversion.

Chapter 5. Using MLE with DFSORT

This chapter provides information on how to use the COBOL SORT and MERGE statements when windowed date fields are used as sort keys. It describes the support given by DFSORT and the restrictions that are imposed, and gives examples.

Notes:

1. This section applies to host platforms only
2. DFSORT is the IBM sort/merge licensed program. Whenever DFSORT is mentioned, any other equivalent sort/merge product can be used.

If your SORT product supports the Y2PAST option and the various windowed year date field identifiers (Y2B, Y2C, Y2D, Y2S, and Y2Z), you can perform sort and merge operations using windowed date fields as sort keys. Virtually all date fields that can be specified with a DATE FORMAT clause are supported, including binary year fields and year-last date fields. The fields will be sorted in windowed year sequence, according to the century window specified by the YEARWINDOW compiler option.

If your SORT product also supports the date field identifiers Y2T, Y2U, Y2W, Y2X, and Y2Y then you can use the TRIG suboption of the DATEPROC compiler option. (Support for these date field identifiers was added to DFSORT/MVS with the application of APAR PQ19684.)

The special indicators that DFSORT recognizes exactly match the triggers and limits supported by COBOL, that is, LOW-VALUE, HIGH-VALUE, and SPACE for alphanumeric date or year fields, and all zeros and all nines for numeric and alphanumeric date fields with at least one non-year digit.

Sorting and Merging

As described in “Sorting and Merging” on page 23, it is possible to use COBOL SORT and MERGE statements to perform sorting operations using windowed date fields as sort keys. The combined software support provided by COBOL and DFSORT will ensure that the records are sorted in windowed year sequence rather than in the sequence of the simple 2-digit number that represents the year.

For a merge operation, the records in the input files must already be in the correct sequence. So where the merge key is a windowed date field, the records must be in windowed date order.

The following sections give information on the restrictions that are imposed with sorting windowed date fields, and show examples of COBOL coding to take advantage of the new capabilities.

For the full syntax of the COBOL SORT and MERGE statements, see the *IBM COBOL Language Reference* for your platform. For a discussion of the sort/merge process and

how to use it in COBOL programs, see the *IBM COBOL Programming Guide* for your platform.

Restrictions with Sorting and Merging

There are some restrictions that you must be aware of when sorting and merging on windowed date fields. This section outlines those restrictions.

Collating Sequence

If you specify an alphanumeric windowed date field as a KEY for a SORT or MERGE statement, then the collating sequence in effect for the sort operation must be EBCDIC. You specify this in one of the following ways:

- In the COLLATING SEQUENCE phrase of the SORT or MERGE statement. This is specified as COLLATING SEQUENCE EBCDIC or COLLATING SEQUENCE NATIVE.
- In the OBJECT-COMPUTER paragraph in the Environment Division. This refers to an *alphabet-name* which is defined in the SPECIAL-NAMES paragraph as NATIVE or EBCDIC.
- By default. EBCDIC is the default collating sequence for all System/390-based programs if neither of the above two options are used.

Output File

When the output from a sort or merge operation is directed to a file (the GIVING phrase), the file cannot be an indexed VSAM file. When an indexed file is initially loaded (by adding records to it), the records must be presented to VSAM in ascending order of the primary key. However, the key sequence that VSAM recognizes is based on the binary value of the key contents, which need not be related to the character set in use, and will probably be different from the sorted value presented by DFSORT when the key is a windowed date field.

The only way to effectively resolve this problem is to expand any date fields used as keys on indexed files. However, this requires that the file format be changed, which in turn will require changes to other programs in the application, and to other control information such as the DEFINE CLUSTER command.

For an example of a program that sorts and expands the key, see “VSAM Key” on page 47.

Examples of Sort Programs

This section shows some examples of COBOL programs that perform sorting operations, where the sort keys are windowed date fields. The examples show how the programs can be changed to take advantage of the millennium language extensions and the Y2PAST option of DFSORT.

The changes in the remediated programs are shown in **bold** type.

Using MLE with DFSORT

Simple Change

The following example shows part of a COBOL program that sorts a transaction file into date and time sequence within account number. In this example, the sort keys are contiguous within the record, and the programmer has taken advantage of this fact to define a single sort key that covers the three keys of account number, date, and time.

```
SD  Sort-File.
    Record Contains 29 Characters
    Data Record is Sort-Record.

01  Sort-Record.
    03  Sort-Key         Pic X(19).       1
    03  Sort-Data        Pic X(10).

FD  Transaction-File
    Record Contains 29 Characters.

01  Transaction-Record.
    03  Trans-Account    Pic 9(8).        2
    03  Trans-Date       Pic 9(5).
    03  Trans-Time       Pic 9(6).
    03  Trans-Type       Pic X.
    03  Trans-Amount     Pic 9(7)V99.

FD  Sorted-File
    Record Contains 29 Characters.
    ⋮
    Sort Sort-File
         On Ascending Key Sort-Key        3
         Using  Transaction-File
         Giving Sorted-File.
```

The following notes apply to this example:

1 Sort-Key, as it is defined in Sort-Record, is defined as the full 19-byte sort sequence rather than the three individual fields. This is a common technique where the sort fields are contiguous, and where the binary sequence equates to the character collating sequence.

2 Transaction-Record maps the same data as Sort-Record, but each field is defined individually for use elsewhere in the program.

3 The SORT statement specifies Sort-Key as the only key for sorting.

To allow DFSORT to treat Trans-Date as a windowed date field, the sort key must be separated into its three component parts, and specified as such in the SORT statement. The following example shows how the program can be changed so that Trans-Date is recognized as a windowed date field.

```
SD  Sort-File.
    Record Contains 29 Characters
    Data Record is Sort-Record.

01  Sort-Record.
    03  Sort-Key.                                          1
        05  Sort-Account Pic 9(8).
        05  Sort-Date    Pic 9(5) Date Format yyxxx.       2
        05  Sort-Time    Pic 9(6).
    03  Sort-Data        Pic X(10).

FD  Transaction-File
    Record Contains 29 Characters.

01  Transaction-Record.
    03  Trans-Account    Pic 9(8).
    03  Trans-Date       Pic 9(5) Date Format yyxxx.       3
    03  Trans-Time       Pic 9(6).
    03  Trans-Type       Pic X.
    03  Trans-Amount     Pic 9(7)V99.

FD  Sorted-File
    Record Contains 29 Characters.
    ⋮
    Sort Sort-File
         On Ascending Key Sort-Account                     4
                          Sort-Date
                          Sort-Time
         Using  Transaction-File
         Giving Sorted-File.
```

The following notes apply to this example:

1. Sort-Key has now been defined with its individual component parts to match the true content of Transaction-Record.
2. Sort-Date is defined as a windowed date field. This instructs COBOL to pass the century window and date format information to DFSORT.
3. Trans-Date is defined as a windowed date field to match Sort-Date.
4. The SORT statement now specifies the three individual sort fields of Account, Date, and Time. Because Sort-Date is a windowed date field, DFSORT will honor the century window in the sort sequence.

VSAM Key

In this example, the sort output is directed to a VSAM indexed file whose key is the date/time stamp of the transaction record. The date is a 6-character Gregorian date in YYMMDD form, and the time is in HHMMSS form.

Using MLE with DFSORT

```
Select T-FILE Assign to Time-Stamp-File
    Organization is Indexed
    Record Key is Time-Stamp-Key.

SD  Sort-File.
    Record Contains 30 Characters
    Data Record is Sort-Record.

01  Sort-Record.
    03  Sort-Account     Pic 9(8).
    05  Sort-Date        Pic 9(6).
    05  Sort-Time        Pic 9(6).
    03  Sort-Data        Pic X(10).

FD  Transaction-File
    Record Contains 30 Characters.

01  Transaction-Record.
    03  Trans-Account    Pic 9(8).
    03  Trans-Date       Pic 9(6).
    03  Trans-Time       Pic 9(6).
    03  Trans-Type       Pic X.
    03  Trans-Amount     Pic 9(7)V99.

FD  Time-Stamp-File
    Record Contains 30 Characters.

01  Time-Stamp-Record.
    03  TS-Account       Pic 9(8).
    03  Time-Stamp-Key.
        05  TS-Date      Pic 9(6).
        05  TS-Time      Pic 9(6).
    03  TS-Type          Pic X.
    03  TS-Amount        Pic 9(7)V99.
    ⋮
    Sort Sort-File
         On Ascending Key Sort-Date
                          Sort-Time
         Using  Transaction-File
         Giving Time-Stamp-File.
```

Because the output file uses a 2-digit year date as part of its key, the data item that represents the date cannot simply be changed to a windowed date field. Although this would cause the records to be output from the sort in their correct logical sequence, the sequence as recognized by VSAM would be incorrect because VSAM uses the binary value of the key. In this case, you need to expand the date and write expanded records to Time-Stamp-File. However, to achieve this, you need to add some program logic at either the sort input or sort output phase. You could use either of the following techniques:

- Write a sort input procedure to read the records from Transaction-File, expand Trans-Date to a 4-digit year, and pass the expanded record to the sort.

- Write a sort output procedure to accept the records from DFSORT in sorted order, expand the date to a 4-digit year, and write the expanded records to Time-Stamp-File.

The following example shows how you could apply the second of these alternatives. An output procedure has been added to the program, and DFSORT now presents the sorted records to the output procedure instead of writing them directly to the VSAM file. The output procedure then expands the date and writes the records to the file.

```
Select T-FILE Assign to Time-Stamp-File
    Organization is Indexed
    Record Key is Time-Stamp-Key.

SD  Sort-File.
    Record Contains 30 Characters
    Data Record is Sort-Record.

01  Sort-Record.
    03  Sort-Account     Pic 9(8).
    05  Sort-Date        Pic 9(6) Date Format yyxxxx.     1
    05  Sort-Time        Pic 9(6).
    03  Sort-Type        Pic X.
    03  Sort-Amount      Pic 9(7)V99.

FD  Transaction-File
    Record Contains 30 Characters.

01  Transaction-Record.
    03  Trans-Account    Pic 9(8).
    03  Trans-Date       Pic 9(6) Date Format yyxxxx.     2
    03  Trans-Time       Pic 9(6).
    03  Trans-Type       Pic X.
    03  Trans-Amount     Pic 9(7)V99.

FD  Time-Stamp-File
    Record Contains 32 Characters.                         3

01  Time-Stamp-Record.
    03  TS-Account       Pic 9(8).
    03  Time-Stamp-Key.
        05  TS-Date      Pic 9(8) Date Format yyyyxxxx.   4
        05  TS-Time      Pic 9(6).
    03  TS-Type          Pic X.
    03  TS-Amount        Pic 9(7)V99.
    ⋮
```

Using MLE with DFSORT

```
       Sort Sort-File
            On Ascending Key Sort-Date
                             Sort-Time
            Using  Transaction-File
            Output Procedure Write-Sorted-Record.          5
       ⋮
   Write-Sorted-Record.                                    6
       Move 0 to End-Sort
       Perform Until End-Sort = 1
          Return Sort-File
             At End
                Move 1 to End-Sort
             Not At-End
                Move Sort-Account to TS-Account
                Move Sort-Date    to TS-Date
                Move Sort-Time    to TS-Time
                Move Sort-Type    to TS-Type
                Move Sort-Amount  to TS-Amount
                Write Time-Stamp-Record
          End-Return
       End-Perform.
```

The following notes apply to this example:

1 Sort-Date is defined as a windowed date field to allow it to be sorted on the correct date sequence.

2 Trans-Date is defined as a windowed date field to match Sort-Date, and for use elsewhere in the program.

3 Because the date field in the output record has been expanded, the size of the record has increased from 30 to 32 characters.

4 TS-Date in Time-Stamp-Record is defined as an expanded date field. This is part of the key of the VSAM file, so it cannot be a windowed date field.

5 The GIVING phrase of the SORT statement has been replaced by the OUTPUT PROCEDURE phrase. This tells DFSORT to pass the records in sorted order to Write-Sorted-Record, instead of writing them directly to Time-Stamp-File.

6 The Write-Sorted-Record procedure copies the sorted records to the output file, expanding the date in the process. It then writes the records to Time-Stamp-File.

SORTING and MERGING with DATEPROC(TRIG)

If the TRIG suboption of DATEPROC is in effect then the SORT or MERGE will check for triggers and limits (or special indicators) in the records and SORT or MERGE accordingly. For numeric windowed date fields are used as ASCENDING/DESCENDING KEYs in SORT or MERGE, all 0's will be treated as lower in the sequence than any other date, and all 9's will be treated as higher than any other date. For alphanumeric windowed date fields, LOW-VALUES, SPACES, and 0's will be treated as lower in the sequence than any other date, and HIGH-VALUES and all 9's will be treated as higher than any other date value.

If your SORT/MERGE KEYs are windowed date fields, and they normally have non-date values in them, such as all 9's or all SPACES, then you will get a different order with SORT and MERGE unless you use the TRIG suboption of the DATEPROC compiler option. With NOTRIG, these values will be interpreted as dates, and windowed accordingly. For example, an alphanumeric date field with DATE FORMAT YYXXXX and a value of all SPACES would be expanded to '19 ' with NOTRIG, but ' ' (all SPACES) with TRIG.

Note: In order to take advantage of the special indicators support in SORT we had to use date formats which also changed the order of the SORT or MERGE for year-last date fields. For example, a date field KEY with DATE FORMAT XXXXYY would be sorted on the XXXX first and then the YY with the NOTRIG option, the same as in any COBOL before. Now with the TRIG option, SORT will sort first on the YY, and then on the XXXX. So, you could get quite different results with TRIG and NOTRIG when sorting or merging with year-last dates. Year-only and year-first dates are not affected in this way.

Other Changes

In the above example, the format of Time-Stamp-File has changed. Both its key length and record length have increased by two bytes. Because of this, you will need to make changes in other parts of the application to match the changes made here. As a guide only, you need to examine the following parts of the application to determine whether they need to be changed:

- Other parts of the same program that reference Time-Stamp-Key or TS-Date
- Other programs in the application suite that read Time-Stamp-File
- The control mechanism, usually either JCL or IDCAMS DEFINE CLUSTER, that creates the VSAM data set to be used by Time-Stamp-File
- Programs in other applications that may read this application's files

Chapter 6. Using MLE with Debug Tool

When you implement date windowing or date expansion using the millennium language extensions, you can use Debug Tool to help test and analyze your modified programs. The following sections provide information to help you use Debug Tool. This information is not a complete guide to using Debug Tool, but should be used in conjunction with, and in addition to, information provided in the *Debug Tool User's Guide and Reference* for your platform.

Preparing to Debug Your Program

Before using Debug Tool you must compile your modified program with the TEST(ALL) or TEST(STMT) compiler option, as well as the DATEPROC option. When you compile with the TEST(ALL) or TEST(STMT) option, the compiler inserts debug hooks in your program at the boundary of each statement, and of each WHEN phrase of an EVALUATE or SEARCH statement. The execution of these debug hooks enables Debug Tool to gain control during program execution. Debug Tool can then take action at different kinds of statements. In particular, with the DATEPROC option, Debug Tool can take action just at date processing statements, that is, those statements that reference date fields.

In addition to specifying the ALL or STMT suboption of the TEST option, if you want to be able to reference the data items in your program (including date fields) by name during your debug session, you must also specify the SYM suboption. To ensure you get all the debugging information you need, specify TEST(ALL,SYM) for the compilation.

For more information about the COBOL TEST compiler option, see the *Debug Tool User's Guide and Reference* for your platform.

Running Your Program with Debug Tool

The best way to run your program with Debug Tool is to use the TEST run-time option to invoke Debug Tool and begin testing your program. For information about invoking Debug Tool, see the *Debug Tool User's Guide and Reference* for your platform.

Using COBOL Date Fields with Debug Tool

As with other program data items, Debug Tool can process valid COBOL date fields. There are, however, a number of restrictions that apply when using date fields in Debug Tool commands:

- Windowing of windowed date fields is not supported. Therefore:
  - Arithmetic operations on windowed date fields are not supported
  - Windowed date fields are not supported in relation conditions

  For more information, see “Evaluation of COBOL Expressions” on page 53.

- The use of windowed date fields as subscripts is not supported.
- Reference modification of date fields is not supported.
- Windowed date fields are not supported as the position or length of a reference-modified data item.
- Arithmetic operations on year-last dates is not supported.

Declaring Temporary Variables

The DATE FORMAT clause is not supported for temporary (session) variables declared during a debug session. Therefore, debug session variables cannot be date fields.

Evaluation of COBOL Expressions

Debug Tool interprets COBOL expressions according to COBOL rules. Some restrictions do apply, and these are described in the *Debug Tool User's Guide and Reference* for your platform. There are also restrictions specific to date fields:

- Windowed date-field operands are not supported in arithmetic expressions in combination with any other operands.

  Examples:

  Assuming the following date fields are defined:

```
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY        PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
```

  The following Debug Tool commands are valid:

```
LIST DATE-YY;

COMPUTE DATE-YY = 97;

COMPUTE DATE-YYYY = DATE-YYYY + 1;
```

  whereas the following Debug Tool commands are invalid:

```
LIST DATE-YY + 1;

COMPUTE DATE-YY = DATE-YY + 1;
```

- Windowed date-field operands are not supported in relation conditions.

  Examples:

  Assuming the following date fields are defined:

```
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY-A      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
77 DATE-YYYY-B      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
```

The following Debug Tool commands are valid:

```
IF DATE-YYYY-A > DATE-YYYY-B THEN ...

IF DATE-YYYY-A > 1900 THEN ...
```

whereas the following Debug Tool commands are invalid:

```
IF DATE-YYYY-A > DATE-YY THEN ...

IF DATE-YY > 99 THEN ...

IF DATE-YY = 99 THEN ...
```

As previously stated, Debug Tool interprets COBOL expressions according to COBOL rules. Normally, Debug Tool does not support Debug Tool commands that would be diagnosed with an error-level message by the COBOL compiler. One exception, however, is the use of expanded date fields in arithmetic expressions. Debug Tool allows expanded date fields to be used anywhere non-dates are allowed in arithmetic expressions.

Debug Tool Commands

The following sections describe the syntax of those Debug Tool commands that provide date processing support.

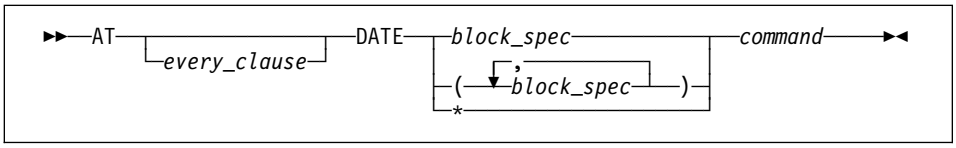
AT Command

The AT command defines a breakpoint or a set of breakpoints. By defining breakpoints you can temporarily suspend program execution and use Debug Tool to perform other tasks.

The following sections describe the forms of the AT command that provide date processing support.

AT DATE

Gives Debug Tool control for each date processing statement within the specified block. The syntax of AT DATE is:



every_clause
A clause that defines how often the specified Debug Tool command is executed. See “Every_Clause” on page 56 for a description of the clause.

block_spec
A valid block specification; see the *Debug Tool User's Guide and Reference* for your platform.

***** Sets a breakpoint at every date processing statement.

command
A valid Debug Tool command.

Usage Notes:

- AT DATE is only supported for compile units compiled with the DATEPROC compiler option.

Examples:

- Each time a date processing statement is encountered in the nested subprogram `subrx`, display the location of the statement.

  ```
  AT DATE subrx QUERY LOCATION;
  ```

- Each time a date processing statement is encountered in the compile unit, display the name of the compile unit.

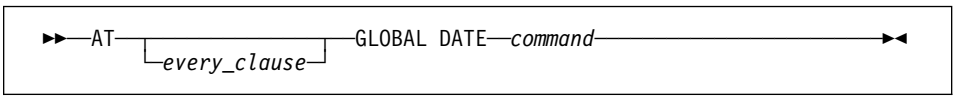
  ```
  AT DATE * LIST %CU;
  ```

- Each time a date processing statement is encountered in the compile unit, display the location of the statement, list a specific variable, and resume running the program.

  ```
  AT DATE * PERFORM
     QUERY LOCATION;
     LIST DATE-FIELD
     GO;
  END-PERFORM;
  ```

AT GLOBAL DATE

AT GLOBAL gives Debug Tool control for every instance of the specified AT-condition. These breakpoints are independent of their non-global counterparts. Global breakpoints are always performed before their specific counterparts. AT GLOBAL DATE gives Debug Tool control for every occurrence of a date processing statement. The syntax of AT GLOBAL DATE is:



every_clause
A clause that defines how often the specified Debug Tool command is executed. See “Every_Clause” on page 56 for a description of the clause.

command
A valid Debug Tool command.

Usage Notes:

- AT GLOBAL DATE is only supported for compile units compiled with the DATEPROC compiler option.

- To set a global breakpoint, you can specify an asterisk (*) as the *every_clause* of the AT command, or you can specify the AT GLOBAL command.

Examples:

- Set a global DATE breakpoint.

  ```
  AT DATE *;
  ```

  or

  ```
  AT GLOBAL DATE;
  ```

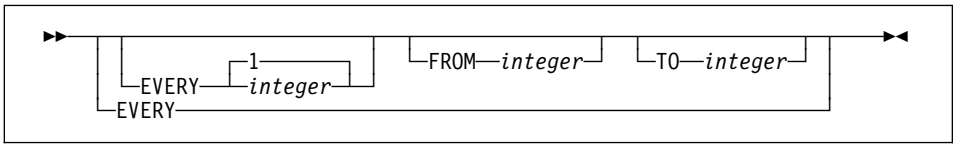
- Every time a date processing statement is encountered, query the location of the statement.

  ```
  AT GLOBAL DATE QUERY LOCATION;
  ```

Every_Clause

Most forms of the AT command contain an optional *every_clause* that controls whether the specified action is taken based on the number of times a situation has occurred. For example, you might want an action to occur only every 10th time a breakpoint is reached.

The syntax for *every_clause* is:



EVERY *integer*
Specifies how frequently the breakpoint is taken. For example, EVERY 5 means that Debug Tool is invoked every fifth time the AT-condition is met. The default is EVERY 1.

FROM *integer*
Specifies when Debug Tool invocations are to begin. For example, FROM 8 means that Debug Tool is not invoked until the eighth time the AT-condition is met. If the FROM value is not specified, its value is equal to the EVERY value.

TO *integer*
Specifies when Debug Tool invocations are to end. For example, TO 20 means that after the 20th time this AT-condition is met, it should no longer invoke Debug Tool. If the TO value is not specified, the *every_clause* continues indefinitely.

Usage Notes:

- FROM *integer* cannot exceed TO *integer* and all integers must be greater than or equal to 1.
- EVERY by itself is the same as EVERY 1 FROM 1.
- The EVERY, FROM, and TO clauses can be specified in any order.

Examples:

- Break every third time statement 50 is reached, beginning with the 48th time and ending after the 59th time. The breakpoint action is performed the 48th, 51st, 54th, and 57th time statement 50 is reached.

  ```
  AT EVERY 3 FROM 48 TO 59 STATEMENT 50;
  ```

- At the fifth change of field `varbl1` print a message saying that it has changed and list its new value. In addition, clear the CHANGE breakpoint.

  ```
  AT FROM 5 CHANGE varbl1 PERFORM
    LIST ("varbl1 has changed. It is now", varbl1):
    CLEAR AT CHANGE varbl1;
  END-PERFORM;
  ```

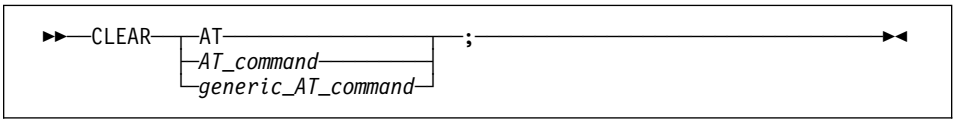
CLEAR Command

The CLEAR command removes the actions of previously issued Debug Tool commands. Some breakpoints are removed automatically when Debug Tool determines that they are no longer meaningful. For example, if you set a breakpoint in a fetched or loaded compile unit, the breakpoint is discarded when the compile unit is released.

The following sections describe the forms of the CLEAR command that provide date processing support.

CLEAR AT

Removes breakpoints from previously issued AT commands (including GLOBAL breakpoints).



AT Removes all breakpoints from previously issued AT commands (including GLOBAL breakpoints).

AT_command
Removes breakpoints for the specified AT command. The AT command specified by *AT_command* must be qualified by at least one operand.

The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted.

generic_AT_command
Removes breakpoints for the specified AT command. The AT command specified by *generic_AT_command* is not qualified by an operand.

Examples:

- Remove the DATE breakpoint for subprogram `subrx`.

  ```
  CLEAR AT DATE subrx;
  ```

- Remove all non-global DATE breakpoints.

  ```
  CLEAR AT DATE;
  ```

- Remove all global DATE breakpoints.

  ```
  CLEAR AT DATE *;
  ```

  or

  ```
  CLEAR AT GLOBAL DATE;
  ```

DESCRIBE Command

The DESCRIBE command displays the attributes of references, compile units, and the execution environment.

DESCRIBE ATTRIBUTES

DESCRIBE ATTRIBUTES displays the attributes of a specified variable. If the variable is defined as a date field, the DATE FORMAT clause associated with the field is displayed (abbreviated to DATEFMT).

Example:

Assuming the following date field is defined:

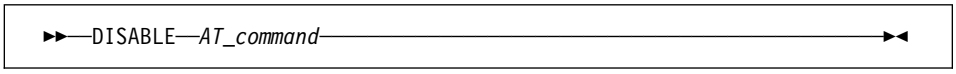
```
77 YYDDD            PIC 9(5) PACKED-DECIMAL
                             DATE FORMAT YYXXX.
```

Then the following would be the output from the DESCRIBE ATTRIBUTES command for the date field:

```
ATTRIBUTES for YYDDD
  Its length is 5
  Its address is 0007C3E0
  01 PGMA:>YYDDD   9(5) DISP DATEFMT YYXXX
```

DISABLE Command

The DISABLE command makes the AT breakpoint inoperative, but does not clear it; you can ENABLE it later without typing the entire command again.



AT_command
Disables the specified AT command. The AT command specified by *AT_command* must be qualified by at least one operand.

The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted.

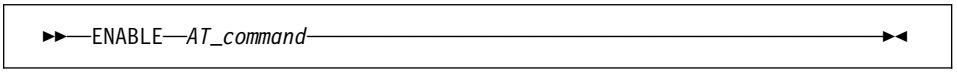
Example:

Disable the breakpoint that was set by the command AT DATE `subrx` QUERY LOCATION;.

```
DISABLE AT DATE subrx;
```

ENABLE Command

The ENABLE command makes the AT breakpoints operative after they have been disabled.



AT_command
Enables the specified AT command. The AT command specified by *AT_command* must be qualified by at least one operand.

The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted.

Example:

Enable the previously disabled AT DATE `subrx` QUERY LOCATION command.

```
ENABLE AT DATE mysub;
```

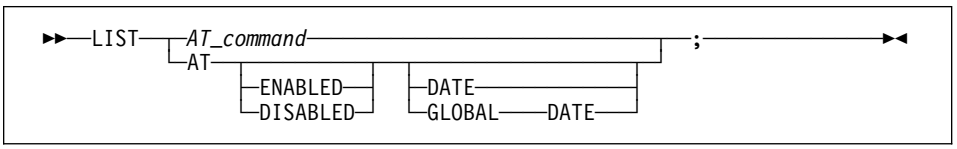
LIST Command

The LIST command displays information about a program such as values of specified variables, structures, arrays, registers, statement numbers, frequency information, and the flow of program execution. The LIST command can be used to display information in any enclave. All information displayed will be saved in the log file.

The following section describes the form of the LIST command that provides date processing support.

LIST AT

Lists the currently defined breakpoints, including the action taken when the specified breakpoint is activated.



AT_command
Lists breakpoint set by the specified AT command. The AT command specified by *AT_command* must be qualified by at least one operand.

The syntax of the AT command must be complete except that the *every_clause* and *command* items are omitted.

ENABLED
Restricts the list to enabled breakpoints. The default is to list both enabled and disabled breakpoints.

DISABLED
Restricts the list to disabled breakpoints. The default is to list both enabled and disabled breakpoints.

DATE
Lists currently defined AT DATE breakpoints.

GLOBAL
Lists currently defined AT GLOBAL breakpoints for the specified AT-condition.

If the AT command type (for example, DATE) is not specified, LIST AT lists all currently defined breakpoints (both disabled and enabled).

Usage Notes:

- LIST AT DATE is only supported for compile units compiled with the DATEPROC compiler option.
- To display a global breakpoint, you can specify an asterisk (*) with the LIST AT command or you can specify the LIST AT GLOBAL command.

Examples:

- List information about global DATE breakpoints.

  ```
  LIST AT DATE *;
  ```

  or

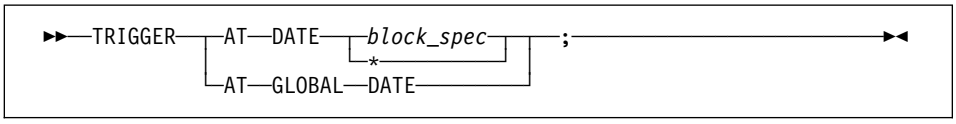
  ```
  LIST AT GLOBAL DATE;
  ```

- List each AT DATE breakpoint in subprogram `subrx`.

  ```
  LIST AT DATE subrx;
  ```

TRIGGER Command

The TRIGGER command raises the specified AT-condition in Debug Tool, or it raises the specified programming language condition in your program. The following section describes the form of the TRIGGER command that provides date processing support. For a complete description of this command, see the *Debug Tool User's Guide and Reference* for your platform.



block_spec
A valid block specification.

Example:

Assuming the following breakpoint exists in a program:

```
AT DATE * LIST "Date processing statement encountered";
```

Trigger the command associated with the AT DATE breakpoint.

```
TRIGGER AT DATE *
```

Considerations for COBOL-Like Commands

The following sections list things you should consider when using date fields in the Debug Tool COBOL-like commands CALL, COMPUTE, EVALUATE, IF, MOVE, and PERFORM.

CALL Command

The CALL command invokes an entry point in the application program. A windowed date field cannot be specified as:

- The identifier containing the entry name.

Examples:

Assuming the following date fields are defined:

```
77 DATE-YYDDD       PIC X(5) DATE FORMAT YYXXX.
77 DATE-YYYYDDD     PIC X(7) DATE FORMAT YYYYXXX.
```

The following Debug Tool commands are valid:

```
CALL "ABC" USING DATE-YYYYDDD
CALL "ABC" USING DATE-YYDDD.
```

COMPUTE Command

The COMPUTE command assigns the value of an arithmetic expression to a specified reference. The following considerations apply to the use of date fields in the COMPUTE command:

- Windowed date fields are not supported in an arithmetic expression if more than one operand is specified. (See “Evaluation of COBOL Expressions” on page 53 for more information.) If a windowed date field is specified as the only operand in an arithmetic expression, the COMPUTE command is treated as a MOVE command, and is subject to the same considerations as the MOVE command (see “MOVE Command” on page 63).
- Year-last date fields are not supported in an arithmetic expression if more than one operand is specified.
- The result of the evaluation of an arithmetic expression containing a year-first expanded date field is considered to be a non-date.

Using MLE with Debug Tool

- Year-first expanded date fields can be used in arithmetic expressions anywhere that non-dates can be used.
- Debug Tool does not support windowing of windowed date fields.

Examples:

Assuming the following date fields are defined:

```
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY-A      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
77 DATE-YYYY-B      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
```

The following Debug Tool commands are valid:

```
COMPUTE DATE-YY = 44

COMPUTE DATE-YYYY-A = 1998

COMPUTE DATE-YYYY-A = DATE-YYYY-A + DATE-YYYY-B

COMPUTE DATE-YYYY-A = DATE-YYYY-A + 1

COMPUTE DATE-YY = DATE-YYYY-A - 1
```

whereas the following Debug Tool commands are invalid:

```
COMPUTE DATE-YYYY-A = DATE-YY

COMPUTE DATE-YY = DATE-YY - 1
```

EVALUATE Command

The EVALUATE command provides a shorthand notation for a series of nested IF statements. See “IF Command” for more information.

IF Command

The IF command lets you conditionally perform a command. Table 2 on page 63 summarizes the permissible comparisons of date fields in relation conditions.

The symbols used in Table 2 on page 63 are:

Y	The relation condition is evaluated using standard COBOL rules.
N	The relation condition is not supported.
Y/N	If the two operands in the relation condition are both expanded date fields with the same DATE FORMAT clause, the relation condition is evaluated using standard COBOL rules. Otherwise, the relation condition is not supported.

Table 2. Permissible Comparisons of Date Fields

	Second Operand		
First Operand	**ND**	**WD**	**XD**
Non-Date (ND)	Y	N	Y
Windowed Date Field (WD)	N	N	N
Expanded Date Field (XD)	Y	N	Y/N

Note: Date fields can be either alphanumeric, external decimal, or internal decimal items, and existing rules for the validity and mode (numeric or non-numeric) of comparing such items still apply.

Examples:

Assuming the following date fields are defined:

```
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY-A      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
77 DATE-YYYY-B      PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
77 DATE-YYYYDDD     PIC 9(7) PACKED-DECIMAL
                             DATE FORMAT YYYYXXX.
```

The following Debug Tool commands are valid:

```
IF DATE-YYYY-A > 1999 THEN ...

IF DATE-YYYY-A > DATE-YYYY-B THEN ...

IF DATE-YYYY-A > DATE-YYYY-B + 1 THEN ...
```

whereas the following Debug Tool commands are invalid:

```
IF DATE-YY > 60 THEN ...

IF DATE-YYYY-A > DATE-YY THEN ...

IF DATE-YYYY-A > DATE-YYYYDDD THEN ...
```

MOVE Command

The MOVE command transfers data from one area of storage to another. Table 3 on page 64 summarizes the behavior of MOVE commands involving date fields.

The symbols used in Table 3 on page 64 are:

Y The MOVE command is performed following the normal COBOL rules.

N The MOVE command is not supported.

Y/N If the sending field and receiving field have the same DATE FORMAT clause, the MOVE command is performed using standard COBOL rules. Otherwise, the MOVE command is not supported.

Table 3. Behavior of Using MOVE Command with Date Fields

	Sending Field		
Receiving Field	**ND**	**WD**	**XD**
Non-Date (ND)	Y	N	Y
Windowed Date Field (WD)	Y	Y/N	N
Expanded Date Field (XD)	Y	N	Y/N

Examples:

Assuming the following fields are defined:

```
77 NON-DATE         PIC 9(4) PACKED-DECIMAL.
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY        PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
77 DATE-YYYYDDD     PIC 9(7) PACKED-DECIMAL
                             DATE FORMAT YYYYXXX.
```

The following Debug Tool commands are valid:

```
MOVE 78 TO DATE-YY;

MOVE 1999 TO DATE-YYYY;

MOVE DATE-YYYY TO NON-DATE;
```

whereas the following Debug Tool commands are invalid:

```
MOVE DATE-YY TO NON-DATE;

MOVE DATE-YY TO DATE-YYYY;

MOVE DATE-YYYY TO DATE-YYYYDDD;
```

PERFORM Command

The PERFORM command transfers control explicitly to one or more statements and implicitly returns control to the next executable statement after execution of the specified statements is completed. The following considerations apply to using date fields in the PERFORM command:

- A windowed date field cannot be specified as an identifier in the VARYING phrase, FROM phrase, or BY phrase of the PERFORM command.
- The PERFORM condition must follow the rules for comparing date fields, as described in “IF Command” on page 62.

Examples:

Assuming the following fields are defined:

```
77 NON-DATE         PIC 9(4) PACKED-DECIMAL.
77 DATE-YY          PIC 9(2) PACKED-DECIMAL
                             DATE FORMAT YY.
77 DATE-YYYY        PIC 9(4) PACKED-DECIMAL
                             DATE FORMAT YYYY.
```

The following Debug Tool command is valid:

```
AT 10 PERFORM WITH TEST BEFORE
   UNTIL DATE-YYYY > 1998
   LIST DATE-YYYY;
END-PERFORM;
```

whereas the following Debug Tool commands are invalid:

```
AT 10 PERFORM WITH TEST BEFORE
   UNTIL DATE-YY > 98
   LIST DATE-YY;
END-PERFORM;

AT 10 PERFORM WITH TEST BEFORE
   VARYING DATE-YY FROM 1 TO 99
   UNTIL NON-DATE = 44
   COMPUTE NON-DATE = NON-DATE + 1;
   LIST NON-DATE;
END-PERFORM;
```

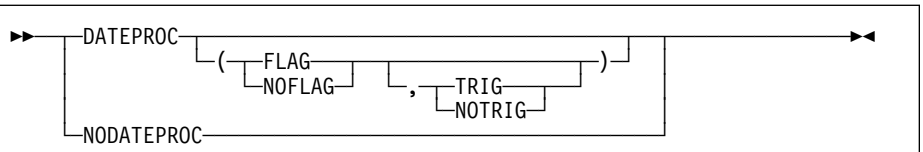
Chapter 7. Millennium Language Extensions Reference

This chapter provides reference information for the millennium language extensions. Information in this chapter is based on the *IBM COBOL Language Reference* and the *IBM COBOL Programming Guide* for the supported platforms. These books contain full syntax descriptions of all of the language constructs and compiler options found here, and you should refer to them for complete details of MLE-related COBOL elements. This chapter contains a summarized presentation of those COBOL elements that are affected by the millennium language extensions.

Compiler Options

The compiler options DATEPROC and YEARWINDOW have been introduced to support the millennium language extensions. For a complete description of all compiler options, see the *IBM COBOL Programming Guide* for your platform.

DATEPROC



Note:
[1] The TRIG/NOTRIG suboption is only applicable to the host.

Default is: NODATEPROC, or DATEPROC(FLAG,NOTRIG) if only DATEPROC is specified

You can specify the FLAG|NOFLAG and TRIG|NOTRIG suboptions in any order. The TRIG|NOTRIG suboption can be specified by itself. If either suboption is omitted, it defaults to the current setting. For information about the order in which compiler options are recognized, see the *IBM COBOL Programming Guide* for your platform.

Abbreviations are: DP|NODP

Use the DATEPROC option to enable the millennium language extensions.

DATEPROC(FLAG)
With DATEPROC(FLAG), the millennium language extensions are enabled, and the compiler will produce a diagnostic message wherever a language element uses or is affected by the extensions. The message will usually be an information-level or warning-level message that identifies statements that involve date-sensitive processing. Additional messages may be generated that identify errors or possible inconsistencies in the date constructs. For information on how to reduce these diagnostic messages, see “Analyzing Date-Related Diagnostic Messages” on page 27.

Production of diagnostic messages, and their appearance in or after the source listing, is subject to the setting of the FLAG compiler option.

DATEPROC(NOFLAG)
With DATEPROC(NOFLAG), the millennium language extensions are in effect, but the compiler will not produce any related messages unless there are errors or inconsistencies in the COBOL source.

DATEPROC(TRIG) - Host Only
With DATEPROC(TRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed date fields is sensitive to specific “trigger” or “limit” values in the date fields and in other non-date fields that are stored into or compared with the windowed date fields. These special values represent invalid dates that can be tested for or used as upper or lower limits. For additional details, see “Triggers, Limits and Special Indicators (Host Only)” on page 20.

DATEPROC(NOTRIG) - Host Only
With DATEPROC(NOTRIG), the millennium language extensions are enabled, and the automatic windowing that the compiler applies to operations on windowed dates does not recognize any special trigger values in the operands. Only the value of the year part of dates is relevant to automatic windowing.

Note: The DATEPROC(NOTRIG) option is a performance option that assumes valid date values in windowed date fields.

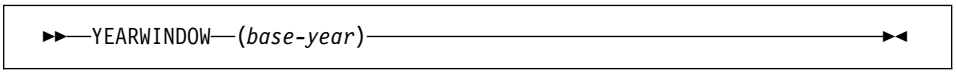
NODATEPROC
NODATEPROC indicates that the extensions are not enabled for this compile unit. This affects date-related program constructs as follows:

- The DATE FORMAT clause is syntax-checked, but has no effect on the execution of the program.
- The DATEVAL and UNDATE intrinsic functions have no effect. That is, the value returned by the intrinsic function is exactly the same as the value of the argument.
- The YEARWINDOW intrinsic function returns a value of zero.

Notes:

1. Specification of the DATEPROC option requires that the NOCMPR2 option is also used.
2. NODATEPROC conforms to the COBOL 85 Standard.

YEARWINDOW



Default is: YEARWINDOW(1900)

Abbreviation is: YW

Use the YEARWINDOW option to specify the first year of the 100-year window (the *century window*) to be applied to windowed date field processing by the COBOL compiler.

base-year represents the first year of the 100-year window, and must be specified as one of the following:

- An unsigned decimal number between 1900 and 1999.

  This specifies the starting year of a fixed window. For example, YEARWINDOW(1930) indicates a century window of 1930–2029.

- A negative integer from -1 through -99.

  This indicates a sliding window, where the first year of the window is calculated from the system date at run time. The number is subtracted from the current year to give the starting year of the century window. For example, YEARWINDOW(-80) indicates that the first year of the century window is 80 years before the current year at the time the program is run.

Notes:

1. The YEARWINDOW option has no effect unless the DATEPROC option is also in effect.
2. At run time, two conditions must be true:
   - The century window must have its beginning year in the 1900s
   - The current year must lie within the century window for the compile unit

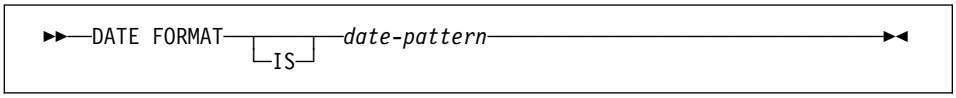
   For example, running a program in 1998 with YEARWINDOW(-99) violates the first condition, and would result in a run-time error.

Data Division Considerations

The following sections outline the various items that can be specified in the Data Division of a COBOL program, and how they affect date fields.

DATE FORMAT Clause

The DATE FORMAT clause specifies that a COBOL data item contains a date in a form supported by the millennium language extensions. This is the primary method you use to specify which date fields are affected by MLE.



The *date-pattern* is a character string representing a windowed (YY) or expanded (YYYY) year. Optionally, it can include up to four characters that represent other parts

of the date. The optional characters can follow the year portion (referred to as a year-first format) or precede the year portion (referred to as a year-last format).

Table 4 lists the acceptable values for the *date-pattern* and the field types that these value represent:

Table 4. Values for date-pattern

Date-pattern string...	Specifies that the data item contains...
YY	A windowed (two-digit) year.
YYYY	An expanded (four-digit) year.
X	A single character. For example, a digit representing a semester or quarter (1-4).
XX	Two characters. For example, digits representing a month (01-12).
XXX	Three characters. For example, digits representing a day of the year (001-366).
XXXX	Four characters. For example, two digits representing a month (01-12). and two digits representing a day of the month (01-31).

The following example shows how you would code the DATE FORMAT clause on a field that contains a date in YYMMDD form.

```
03 Invoice-Date     Pic 9(6) Date Format yyxxxx.
```

Level Numbers

A date field can be defined with any level number from 01 to 49, or level 77. It can be a group item or an elementary item.

Date fields can also have level-88 items associated with them (see “VALUE Clause” on page 71), and level-66 items (see “RENAMES Clause” on page 73).

A group item can be defined as a date field provided that its subordinate items match the definition of the group item. For example, to define a windowed Gregorian date field and subdivide it into its date components, you could specify:

```
03  DATE-FIELD          DATE FORMAT YYXXXX.
    05  YEAR-PART       PIC 99 DATE FORMAT YY.
    05  MONTH-PART      PIC 99.
    05  DAY-PART        PIC 99.
```

In this way, you can use DATE-FIELD or any of its subordinates as a single data item. Because both DATE-FIELD and YEAR-PART have a DATE FORMAT clause, the century window will be applied to all references to ensure that the date is consistent.

Note that, because DATE-FIELD has a DATE FORMAT clause, YEAR-PART should also be given one as in this example. If it is not, it will be treated as a non-date, and any changes to the contents of YEAR-PART will not be windowed correctly. This will

make the contents of DATE-FIELD unreliable. However, if the subordinate data item that matches the year is not named (or is defined as FILLER), it does not need a DATE FORMAT clause because it cannot be referenced separately.

With Gregorian dates, you can also specify the year and month together as a single data item, which is subordinate to the entire date field. For example:

```
03  DATE-FIELD          DATE FORMAT YYXXXX.
    05  YEAR-AND-MONTH  PIC 9999 DATE FORMAT YYXX.
    05  DAY-PART        PIC 99.
```

You can also define a windowed date field as a subordinate item to an expanded date field, provided the last two characters of the year are in the same location. For example:

```
03  EXPANDED-DATE       DATE FORMAT YYYYXXXX.
    05  CENTURY-PART    PIC XX.
    05  WINDOWED-DATE   PIC X(6) DATE FORMAT YYXXXX.
```

You must be sure not to split the last two characters of the year component when you define subordinate date fields. For example, with the following description:

```
01  DATE-FIELD-2        DATE FORMAT YYXXXX.
    03  YEAR-FIRST-DIGIT          PIC 9.
    03  DATE-WITH-ONE-DIGIT-YEAR  PIC 9(5).
```

the compiler is unable to maintain consistency between the year component of DATE-FIELD-2 and its two subordinate data items, because the three fields can be referenced at different places in the program.

USAGE Clause

The USAGE clause defines the internal representation of the data in the field. For date fields, the USAGE clause can only specify one of the following data types:

BINARY
COMPUTATIONAL
COMPUTATIONAL-3
COMPUTATIONAL-4
DISPLAY
PACKED-DECIMAL

The DATE FORMAT clause is not allowed for USAGE COMP data items if the TRUNC(BIN) compiler option is in effect.

In this context, COMP-3, COMPUTATIONAL-3, and PACKED-DECIMAL are considered to be identical.

With alphanumeric data items (USAGE DISPLAY and PICTURE X), it is possible for the field to contain non-numeric data, even if it is defined as a date field. Or the part of the field corresponding to the year may contain numeric data, but not the remainder of the field. For example, a field defined as:

```
01  YEAR-MONTH          PIC X(5) DATE FORMAT YYXXX.
```

could contain “98Jan”, “00***”, “*****”, or HIGH-VALUES.

The DATE FORMAT clause indicates that if the year part of the field contains numeric data, it should be treated as a windowed year. This may cause some changes to the semantics of your program. For example, in the original program, “00***” was not intended to represent a date; it signifies a control record in the input file. However, because it is a date field, it will be treated as 1900 or 2000 depending on the century window. For more information, see “UNDATE” on page 26.

| **Note:** On the host, if the DATEPROC(TRIG) compiler option is in effect, the comparison is sensitive to date trigger values (such as LOW-VALUE, HIGH-VALUE, or SPACE) in the non-date operand. For more information, see “DATEPROC” on page 66.

VALUE Clause

Expanded date fields are not affected by the century window specified for the compile unit. The year in an expanded date field is always relative to zero (virtual year 0 AD). Therefore, if an expanded date field is defined with a VALUE clause, the value specified in the clause is treated as a non-date literal, and the compiler does not attempt to interpret the value according to the century window. For example, in the following definition, the compiled value of DATE-FIELD-Y4 will always be 00980520 regardless of the century window:

```
01  DATE-FIELD-Y4       PIC 9(8) DATE FORMAT YYYYXXXX
                                 VALUE 980520.
```

However, with a windowed date field, the year is based on the century window. In the following example, DATE-FIELD-Y2 is a windowed date field because it uses a 2-digit year. If the century window is 1950–2049, the value of DATE-FIELD-Y2 is processed as 19980520.

```
05  DATE-FIELD-Y2   PIC 9(6) DATE FORMAT YYXXXX
                             VALUE 980520.
```

Level 88 Condition-Name

The VALUE clause of a condition-name associated with a date field is similar to the VALUE clause discussed above. For an expanded date field, the VALUE clause of a condition-name represents the full 4-digit year; the century window has no effect. For a windowed date field, it represents a year based on the century window.

| For non-year-last dates, if a range of values (*literal-1* THRU *literal-2*) is specified for a condition-name associated with a windowed date field, then the windowed value (not the face value) of *literal-2* must be greater than that of *literal-1*. For example, if the century window were 1990–2089, then the following specification would be valid:

```
05  YEAR-FIELD      PICTURE 99 DATE FORMAT YY.
88  YEAR-RANGE                VALUE 98 THRU 06.
```

However, the compiler can only verify this if the YEARWINDOW compiler option specifies a fixed century window (for example, YEARWINDOW(1940) rather than YEARWINDOW(-60)).

For this reason, if the YEARWINDOW compiler option specifies a sliding century window, you cannot use the THRU option on the VALUE clause of a level-88 condition name.

Note: The windowed order requirement does not apply to year-last date fields. If you specify a condition-name VALUE clause with the THROUGH phrase for a year-last date field, the two literals must follow normal COBOL rules. That is, the first literal must be less than the second literal.

REDEFINES Clause

The REDEFINES clause provides a means for defining an alias for a data item, where both the original data item and the alias can be acted on by statements in the program. This, and the RENAMES clause, provide an alias facility that the compiler can recognize. That is, the compiler "knows" that both the original field and the redefining field occupy the same storage, and that an operation on one field affects the other.

This ensures that date fields are treated consistently throughout the compile unit.

Notes:

1. For the purpose of this discussion, where there are multiple level-01 record description entries for a File Description, this is equivalent to using the REDEFINES clause in the WORKING-STORAGE section, and the same considerations apply.
2. There are also ways in which a date field can have an alias without it being visible to the compiler. See "Hidden Aliases" on page 74 for details.

The following example shows a windowed Gregorian date field specified as a single alphanumeric field:

```
01  DATE-YYMMDD         PIC X(6) DATE FORMAT YYXXXX.
```

This can be redefined as a numeric field as follows:

```
01  DATE-YYMMDD-E       PIC 9(6) REDEFINES DATE-YYMMDD
                                  DATE FORMAT YYXXXX.
```

or as a group item with subordinate items:

```
01  DATE-YYMMDD-GRP     REDEFINES DATE-YYMMDD
                         DATE FORMAT YYXXXX.
    03  YY-1            PIC XX   DATE FORMAT YY.
    03  MMDD-1          PIC X(4).
    03  MMDD-GRP-1      REDEFINES MMDD-1.
        05  MM-1        PIC XX.
        05  DD-1        PIC XX.
```

or as a group item with the month component isolated:

```
01  DATE-YYMMDD-GRP2    REDEFINES DATE-YYMMDD.
    03                  PIC XX.
    03  MM-2            PIC 99.
    03                  PIC XX.
```

With Gregorian dates, you can also specify the year and month together as a single data item that redefines the matching part of the full date field. For example:

```
01  DATE-YYMMDD         PIC X(6) DATE FORMAT YYXXXX.
01  DATE-YYMM           REDEFINES DATE-YYMMDD
                        PIC X(4) DATE FORMAT YYXX.
```

Diagnostic Messages

When you use the REDEFINES clause in conjunction with date fields, the compiler will generate warning diagnostics if the redefining field does not have DATE FORMAT clauses consistent with the redefined field. For example, with the following code:

```
01  DATE-YYMMDD         PIC X(6) DATE FORMAT YYXXXX.
01  DATE-YY             REDEFINES DATE-YYMMDD
                        PIC XX.
```

a warning message will be produced indicating that the field DATE-YY should be specified as DATE FORMAT YY, to match that specified in the redefined field. You should examine all such warning messages carefully to determine whether or not this is an error. For more information, see “Analyzing Date-Related Diagnostic Messages” on page 27.

RENAMES Clause

In addition to the REDEFINES clause, the RENAMES clause also provides a means for defining an alias for a data item, where both the original data item and the alias can be acted on by statements in the program. However, in contrast to REDEFINES, RENAMES does not allow you to specify DATE FORMAT for the new name. This means that when you refer to the item by its new name (its alias), COBOL is unable to maintain the correct date windowing facilities for the field.

For this reason, the compiler produces a warning-level diagnostic message whenever a RENAMES clause covers all or part of the year component of a windowed date field.

The following example shows a group item windowed date field as part of a larger group item:

```
01  A-RECORD.
    03  OTHER-FIELD     PIC X(10).
    03  DATE-FIELD      DATE FORMAT YYXXXX.
        05  DATE-YY     PIC XX DATE FORMAT YY.
        05  DATE-MM     PIC XX.
        05  DATE-DD     PIC XX.
    03  TIME-FIELD.
        05  TIME-HH     PIC XX.
        05  TIME-MM     PIC XX.
        05  TIME-SS     PIC XX.
```

The following examples of the RENAMES clause can be used to redefine parts of this structure:

```
66  DATE-TIME-STAMP     RENAMES DATE-FIELD THROUGH TIME-FIELD.
66  DATE-STAMP          RENAMES DATE-FIELD.
66  MONTH-AND-DAY       RENAMES DATE-MM THROUGH DATE-DD.
```

In these examples:

- The definition of DATE-TIME-STAMP causes a warning message to be produced, and you should check all occurrences of DATE-TIME-STAMP in the body of the program to determine whether it will interfere with the integrity of the date.
- The definition of DATE-STAMP also causes a warning message. This is more likely to cause a date windowing problem than DATE-TIME-STAMP because it exactly matches DATE-FIELD, which is a windowed Gregorian date field, and is therefore likely to be used in the body of the program as a date field in its own right.
- The definition of MONTH-AND-DAY is accepted because it does not overlap the year component of the date field.

Hidden Aliases

As described in “REDEFINES Clause” on page 72, it is possible for a windowed or expanded date field to have one or more aliases which the compiler recognizes because of the REDEFINES or RENAMES clause. However, it is also possible for a date field to have an alias that is invisible to the compiler.

Some ways that this might happen are:

- More than one program can process the same file, but use different record descriptions. One program defines the field as a date field and writes records containing that field to the file. Then another program reads the file but does not specify that the field is a windowed or expanded date field.
- A program CALLs a subroutine, passing as one of its arguments a group item containing a date field, but the called routine does not define the field in the matching parameter as a date field.
- A program defines two or more different record layouts based on the same POINTER, and one of these records contains a date field. The other records occupy the same storage, but are treated differently because they do not contain date fields.
- A CICS program does an EXEC CICS LINK to another program and passes a COMMAREA. The COMMAREA contents are described as a level-01 group item that contains a windowed or expanded date field, but the matching definition in the linked-to program does not define it as a date field.

Procedure Division

The millennium language extensions provide some enhancements to COBOL Procedure Division statements to recognize and act on windowed and expanded date fields. This section outlines the new facilities that are available.

MOVE Statement

MOVEs involving year-last dates are performed as if all items were non-dates. Both the sending item and receiving field must have identical date formats. When a year-last date field is specified as a receiving item, then the sending item can be either a year-last date or a non-date.

For MOVEs involving non-year last dates, you can convert windowed date fields to expanded date fields using simple COBOL statements such as MOVE. For example, in the following code:

```
01  DATE-SHORT          PIC X(6) DATE FORMAT YYXXXX.
01  DATE-LONG           PIC X(8) DATE FORMAT YYYYXXXX.
    ⋮
    MOVE DATE-SHORT TO DATE-LONG.
```

if DATE-SHORT contains the value 981105 and the century window is 1950–2049, the MOVE statement converts the windowed date to its expanded form using the century window, to give the result 19981105.

You can also move expanded date fields to windowed date fields, as follows:

Numeric move
The data is right-justified in the receiving field, and the leading two digits of the year are truncated. Note that, if the date in the sending field is outside the century window, the results will be incorrect. If this is a possibility, you could use a COMPUTE statement with a SIZE ERROR phrase, instead of a MOVE statement. For more information, see “ON SIZE ERROR Phrase” on page 79.

Alphanumeric move
The data is right-justified in the receiving field, and the leading two characters of the year are truncated (note that this does not follow the normal COBOL convention of left-justified alphanumeric moves). If the date in the sending field is outside the century window, the results will be incorrect. For more information, see “Contracting Moves” on page 29.

You can move a non-date to a date field, but you must ensure that the date information in the non-date is compatible with the DATE FORMAT specified for the receiving field. For example:

```
*    Following field contains a date in YYMMDD form
 01  INPUT-DATE          PIC X(6).
 01  RECEIVING-DATE      PIC X(6) DATE FORMAT YYXXXX.
     ⋮
     MOVE INPUT-DATE TO RECEIVING-DATE.
```

The compiler produces a warning-level diagnostic message telling you that a non-date is being moved to a windowed date field.

You cannot move a date field to a non-date. However, you can use the UNDATE intrinsic function to achieve such a move. For information on the UNDATE intrinsic function, see “UNDATE Intrinsic Function” on page 84.

Input/Output Statements

You can also use some of the standard COBOL input/output statements to convert windowed date fields to expanded date fields; for example, the INTO option of the READ statement. If the sending field (defined in the File Description entry) consists solely of a windowed date field, and the receiving field (specified with the INTO option) is an expanded date field, then the READ statement expands the date as part of its implied MOVE processing. For example:

```
FD  DATE-PARM-FILE
    RECORDING MODE IS F
    RECORD CONTAINS 6 CHARACTERS.
01  INPUT-DATE          PIC 9(6) DATE FORMAT YYXXXX.
    ⋮
WORKING-STORAGE SECTION.
01  INPUT-DATE-E        PIC 9(8) DATE FORMAT YYYYXXXX.
    ⋮
PROCEDURE DIVISION.
    READ DATE-PARM-FILE INTO INPUT-DATE-E.
```

In this example, the READ statement performs an implied MOVE to copy INPUT-DATE to INPUT-DATE-E. This expands the date using the century window, as described in the MOVE statement above.

Other input/output operations that perform this date expansion are:

- The FROM phrase of a REWRITE statement
- The FROM phrase of a WRITE statement
- The INTO phrase of a RETURN statement

Date Comparisons

The millennium language extensions allow you to compare two windowed date fields and get results that are consistent with the century window. For example, consider the following code when the century window is 1950–2049:

```
01  THIS-YEAR           PIC XX DATE FORMAT YY.
01  LAST-YEAR           PIC XX DATE FORMAT YY.
    ⋮
    IF THIS-YEAR GREATER THAN LAST-YEAR ...
```

If THIS-YEAR contains the value 00 and LAST-YEAR contains 99, then the comparison evaluates to TRUE, because the comparison is computed as if it were:

```
    IF 2000 GREATER THAN 1999 ...
```

Similarly, if THIS-YEAR contains 50 and LAST-YEAR contains 49, the comparison evaluates to FALSE, because the comparison is computed as if it were:

```
    IF 1950 GREATER THAN 2049 ...
```

You can also compare windowed date fields with expanded date fields and get correct results. For example:

```
01  JDATE-SHORT         PIC X(5) DATE FORMAT YYXXX.
01  JDATE-LONG          PIC X(7) DATE FORMAT YYYYXXX.
    ⋮
    IF JDATE-SHORT = JDATE-LONG ...
```

In this example, if JDATE-SHORT contains 98107 and JDATE-LONG contains 1998107, and the century window is set such that the windowed year 98 is actually 1998, then the comparison evaluates to TRUE.

Literals and Non-Dates

You can compare a non-date with a date field, but you must ensure that the date information in the non-date is compatible with the DATE FORMAT specified for the date field. The compiler assumes that the date formats are the same. For example:

```
*    Following field contains a date in YYMMDD form
 01  DELIVERY-DATE       PIC X(6).
 01  RECEIVED-DATE       PIC X(6) DATE FORMAT YYXXXX.
     ⋮
     IF DELIVERY-DATE = RECEIVED-DATE ...
```

The compiler interprets the non-date comparand as a compatible windowed date field, and produces a warning-level diagnostic message to inform you of this. However, because DELIVERY-DATE is a non-date, the window used to expand it is the assumed window of 1900–1999. This may be different from the window used to expand RECEIVED-DATE, which is the century window for the compile unit. For more information, see “Treatment of Non-Dates” on page 31.

Arithmetic Expressions

The millennium language extensions allow arithmetic operations on numeric date fields in the same manner as any numeric data item. Where the date field is a windowed date, the century window will be used in the calculation. However, there are some restrictions on where date fields can be used in arithmetic expressions.

Arithmetic operations that include date fields are restricted to:

- Adding a non-date to a date field
- Subtracting a non-date from a date field
- Subtracting a date field from a compatible date field to give a non-date result

The following arithmetic operations are not allowed:

- Any operation between incompatible date fields
- Adding two date fields
- Subtracting a date field from a non-date
- Unary minus, applied to a date field
- Multiplication, division, or exponentiation of or by a date field
- Arithmetic expressions that specify a year-last date field.

- Arithmetic statements that specify a year-last date field, except as a receiving data item when the sending field is a non-date.

Windowed Date Fields

Where a windowed date field participates in an arithmetic operation, it is automatically expanded according to the century window specified by the YEARWINDOW compiler option. For the host, when DATEPROC(TRIG) is in effect, this expansion is sensitive to trigger values in the date field. For example:

```
01  Review-Record.
    03  Last-Review-Year    Pic 99 Date Format yy.
    03  Next-Review-Year    Pic 99 Date Format yy.
    ⋮
    Add 10 to Last-Review-Year Giving Next-Review-Year.
```

If the century window is 1910–2009, and the value of Last-Review-Year is 98, then the computation proceeds as if Last-Review-Year is first incremented by 1900 to give 1998. Then the ADD operation is performed, giving a result of 2008. This is stored in Next-Review-Year as 08.

Signed Date fields: If a windowed date field is signed, it is possible for a negative number to be treated as a valid date. In the above example, the determination of the increment (1900 or 2000) to be applied to Last-Review-Year is based on the last two digits of the starting year of the century window (10 in this case). If the value of Last-Review-Year is less than 10, the increment is 2000, otherwise the increment is 1900.

For example, if Last-Review-Year is declared as:

```
    03  Last-Review-Year    Pic S99 Date Format yy.
```

it can contain a negative number because it is a signed field. If, for example, it contains -2, this is less than 10, so an increment of 2000 is applied. This gives a result of 1998, so the value of Last-Review-Year is considered to be 1998.

Note that this can also result in a year that is outside the century window. If Last-Review-Year contains -95, this is also less than 10, so an increment of 2000 gives a result of 1905. The value of Last-Review-Year is considered to be 1905 in this case.

Order of Evaluation

Because of the restrictions on date fields in arithmetic expressions, you may find that programs that previously compiled successfully now produce diagnostic messages when some of the data items are changed to date fields.

Consider the following example:

```
01  Dates-Record.
    03  Start-Year-1        Pic 99 Date Format yy.
    03  End-Year-1          Pic 99 Date Format yy.
    03  Start-Year-2        Pic 99 Date Format yy.
    03  End-Year-2          Pic 99 Date Format yy.
    ⋮
    Compute End-Year-2 = Start-Year-2 + End-Year-1 - Start-Year-1.
```

In this example, the first arithmetic expression evaluated is:

```
Start-Year-2 + End-Year-1
```

However, this is the addition of two date fields, which is not permitted. To resolve this, you should use parentheses to isolate those parts of the arithmetic expression that are allowed. For example:

```
Compute End-Year-2 = Start-Year-2 + (End-Year-1 - Start-Year-1).
```

In this case, the first arithmetic expression evaluated is:

```
End-Year-1 - Start-Year-1
```

This is the subtraction of one date field from another, which is permitted, and gives a non-date result. This non-date result is then added to the date field End-Year-1, giving a date field result which is stored in End-Year-2.

ON SIZE ERROR Phrase

In the example in “Windowed Date Fields” on page 78, the result of 2008 falls within the century window of 1910–2009, so a value of 08 in Next-Review-Year will be recognized as 2008 by subsequent statements in the program.

However, the statement:

```
Add 20 to Last-Review-Year Giving Next-Review-Year.
```

would give a result of 2018. As this falls outside the range of the century window, if the result is stored in Next-Review-Year it would be incorrect, because later references to Next-Review-Year would interpret it as 1918. In this case, the result of the operation depends on whether the ON SIZE ERROR phrase is specified on the ADD statement, as follows:

- If SIZE ERROR is specified, the receiving field is not changed, and the SIZE ERROR imperative statement is executed.
- If SIZE ERROR is not specified, the result is stored in the receiving field with the left-hand digits truncated.

This is an important consideration when developing an internal bridging solution to resolve a date processing problem (see “Internal Bridging” on page 15). When you contract a 4-digit year date field back to 2 digits to write it to the output file, you need to ensure that the date falls within the century window, and that therefore the 2-digit year date will be represented correctly in the field.

You can achieve this using a COMPUTE statement to do the contraction, with a SIZE ERROR phrase to handle the out-of-window condition. For example:

```
Compute Output-Date-YY = Work-Date-YYYY
        On Size Error Go To Out-of-Window-Error-Proc.
```

ON SIZE ERROR Processing and Trigger Values (Host Only): If the DATEPROC(TRIG) compiler option is in effect, and the sending field contains a trigger (either zero or all nines) the size error imperative statement is executed. The result is not stored in the receiving field.

A non-date is considered to have a trigger value of all nines if it has a nine in every digit position of its assumed date format. Thus, for a receiving date format of YYXXX, the non-date value of 99,999 is a trigger, but the values 9,999 and 999,999 are not, although the larger value of 999,999 will cause a size error anyway.

Arithmetic Semantics

All arithmetic, whether performed on date fields or not, acts only on the numeric contents of the fields; date semantics other than expanding and contracting the year part of the date as appropriate for the other operands in the statement are not recognized. For example, adding 1 to a windowed Gregorian date field that contains the value 980831 gives a result of 980832, not 980901.

ACCEPT Statement

There are new implications for the Format 2 ACCEPT statement, when it is used to receive the system date in Gregorian or Julian form. When the DATEPROC compiler option is in effect, the data item into which the date is accepted *must* have a DATE FORMAT clause, and its date format must match the date type specified in the ACCEPT statement.

This applies to the following forms of the ACCEPT statement:

ACCEPT *identifier* FROM DATE — Windowed Gregorian date (YYXXXX)

ACCEPT *identifier* FROM DAY — Windowed Julian date (YYXXX)

ACCEPT *identifier* FROM DATE YYYYMMDD — Expanded Gregorian date (YYYYXXXX)

ACCEPT *identifier* FROM DAY YYYYDDD — Expanded Julian date (YYYYXXX)

This is an important consideration when you change your COBOL programs to take advantage of the millennium language extensions. If *identifier* does not have a DATE FORMAT clause, the compiler produces an error-level diagnostic message for the ACCEPT statement.

Intrinsic Functions

The millennium language extensions include three intrinsic functions to assist with manipulating and converting date fields. These functions are:

DATEVAL Converts a non-date to a date field

UNDATE Converts a date field to a non-date

YEARWINDOW Interrogates the century window

Other intrinsic functions are available in COBOL to convert 2-digit year date fields to expanded date fields. These functions are:

DATE-TO-YYYYMMDD
Expands a 2-digit year Gregorian date to a date with a 4-digit year

DAY-TO-YYYYDDD Expands a 2-digit year Julian date to a date with a 4-digit year

YEAR-TO-YYYY Expands a 2-digit year to a 4-digit year

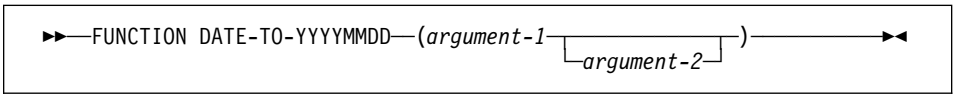
These six intrinsic functions are described below.

DATE-TO-YYYYMMDD Intrinsic Function

The DATE-TO-YYYYMMDD function converts argument-1 from a date with a 2-digit year (YYnnnn) to a date with a 4-digit year (YYYYnnnn). Argument-2, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

If the DATEPROC compiler option is in effect, then the returned value is an expanded date field with implicit date format YYYYXXXX.



argument-1
Must be zero or a positive integer less than 1,000,000.

argument-2
Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

Example

Some examples of returned values from the DATE-TO-YYYYMMDD function follow:

Current Year	Argument-1 Value	Argument-2 Value	DATE-TO-YYYYMMDD Return Value
2002	851003	120	20851003
2002	851003	-20	18851003
2002	851003	10	19851003
1994	981002	-10	18981002

DATEVAL Intrinsic Function

The DATEVAL function converts a non-date to a date field, for unambiguous use with date fields.

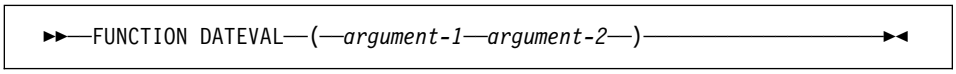
If the DATEPROC compiler option is in effect, the returned value is a date field containing the value of argument-1 unchanged. For information on using the resulting date field:

- In arithmetic, see “Arithmetic Expressions” on page 77.
- In conditional expressions, see “Date Comparisons” on page 18.

If the NODATEPROC compiler option is in effect, the DATEVAL function has no effect, and returns the value of argument-1 unchanged.

The function type depends on the type of argument-1:

Argument-1 Type	Function Type
Alphanumeric	Alphanumeric
Integer	Integer



argument-1
Must be one of the following:

- A class alphanumeric item with the same number of characters as the date format specified by argument-2.
- An integer. This can be used to specify values outside the range specified by argument-2, including negative values.

The value of argument-1 represents a date of the form specified by argument-2.

argument-2
Must be a nonnumeric literal specifying a date pattern, as defined in “DATE FORMAT Clause” on page 68. The date pattern consists of YY (representing a windowed year) or YYYY (representing an expanded year). Optionally, it can include up to four characters that represent other parts of the date. The optional characters can follow the year portion (referred to as a year-first format) or precede the year portion (referred to as a year-last format).

Table 5 lists the acceptable values for *argument-2* and the field types that these value represent:

argument-2 values

Table 5 (Page 1 of 2).

Date-pattern string for *argument-2*...	**Specifies that argument-1 contains...**
YY	A windowed (two-digit) year.
YYYY	An expanded (four-digit) year.
X	A single character. For example, a digit representing a semester or quarter (1-4).

Table 5 (Page 2 of 2).

Date-pattern string for *argument-2*...	Specifies that argument-1 contains...
XX	Two characters. For example, digits representing a month (01-12).
XXX	Three characters. For example, digits representing a day of the year (001-366).
XXXX	Four characters. For example, two digits representing a month (01-12). and two digits representing a day of the month (01-31).

Note: You can use apostrophes as the literal delimiters instead of quotes (independent of the APOST/QUOTE compiler option).

Examples

The following example shows a call to DATEVAL to convert an alphanumeric field to an expanded date field:

```
01  INPUT-DATE          PIC X(6).
01  OUTPUT-DATE         PIC X(8) DATE FORMAT YYYYXXXX.
    ⋮
    MOVE FUNCTION DATEVAL (INPUT-DATE "YYXXXX") TO OUTPUT-DATE.
```

The following example shows an incorrect date comparison, and the proper use of the DATEVAL intrinsic function to correct the comparison.

```
01  DATE-1              PIC 9(6) DATE FORMAT YYXXXX.
    ⋮
    IF DATE-1 = 021016 ...

    IF DATE-1 = FUNCTION DATEVAL (021016, "YYXXXX") ...
```

In this example, the first comparison is incorrect because the compiler expands the two dates using different century windows. The value in DATE-1 is expanded using the century window for the compile unit. So, if it contains 021016, and the century window is 1920–2019, then it is treated as 20021016. But the constant 021016 is windowed against the assumed century window of 1900–1999, so it is treated as 19021016.

In the second comparison, the DATEVAL intrinsic function converts the constant to a windowed date field, and it is this windowed date field that takes part in the comparison. It is expanded using the century window for the compile unit, so this ensures that the comparison is consistent.

For more information on the behavior of literals, see “Treatment of Non-Dates” on page 31.

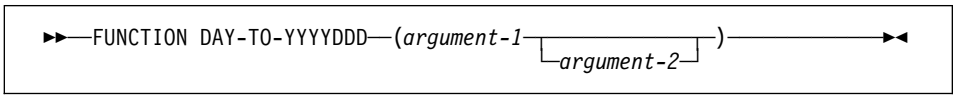
DAY-TO-YYYYDDD Intrinsic Function

The DAY-TO-YYYYDDD function converts argument-1 from a date with a 2-digit year (YYnnn) to a date with a 4-digit year (YYYYnnn). Argument-2, when added to the year

at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

If the DATEPROC compiler option is in effect, then the returned value is an expanded date field with implicit date format YYYYXXX.



argument-1
Must be zero or a positive integer less than 100,000.

argument-2
Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

Example

Some examples of returned values from the DAY-TO-YYYYDDD function follow:

Current Year	Argument-1 Value	Argument-2 Value	DAY-TO-YYYYDDD Return Value
2002	10004	-20	1910004
2002	10004	-120	1810004
2002	10004	20	2010004
2013	95005	-10	1995005

UNDATE Intrinsic Function

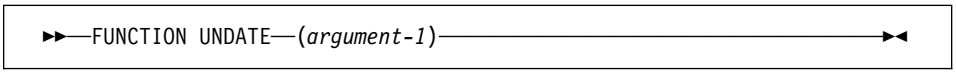
Warning: The use of UNDATE should be avoided, except as a last resort, since the compiler will 'lose' the flow of date fields in your program. This could result in date comparisons not being windowed properly. Use more DATE FORMAT clauses instead of function UNDATE for MOVE and COMPUTE.

The UNDATE function converts a date field to a non-date for unambiguous use with non-dates.

If the NODATEPROC compiler option is in effect, the UNDATE function has no effect.

The function type depends on the type of argument-1:

Argument-1 Type	Function Type
Alphanumeric	Alphanumeric
Integer	Integer



argument-1
A date field.

The returned value is a non-date that contains the value of argument-1 unchanged.

Examples

The following example shows a call to UNDATE to convert a date field to a non-date:

```
01  INPUT-DATE          PIC X(6) DATE FORMAT YYXXXX.
01  OUTPUT-DATE         PIC X(6).
    ⋮
    MOVE FUNCTION UNDATE (INPUT-DATE) TO OUTPUT-DATE.
```

One reason you might want to "undate" a date is to use it as an argument to an intrinsic function, such as DATE-TO-YYYYMMDD. Normally, windowed date fields are not permitted as arguments to intrinsic functions (except, of course, the UNDATE function itself). The following example shows how you can use UNDATE to convert a windowed date field to a non-date, and use it in a call to the DATE-TO-YYYYMMDD intrinsic function:

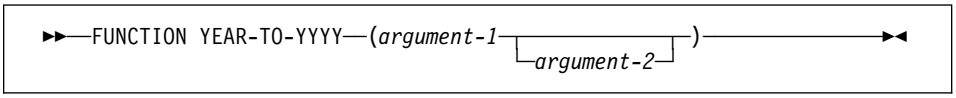
```
01  FROM-DATE           PIC 9(6) DATE FORMAT YYMMDD.
01  TO-DATE             PIC 9(8).
01  F-WINDOW            PIC 9(8) BINARY.
    ⋮
    MOVE FUNCTION DATE-TO-YYYYMMDD (FUNCTION UNDATE (FROM-DATE) F-WINDOW)
         TO TO-DATE.
```

YEAR-TO-YYYY Intrinsic Function

The YEAR-TO-YYYY function converts argument-1, a 2-digit year, to a 4-digit year. Argument-2, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of argument-1 falls.

The function type is integer.

If the DATEPROC compiler option is in effect, then the returned value is an expanded date field with implicit date format YYYY.



argument-1
Must be a non-negative integer that is less than 100.

argument-2
Must be an integer. If argument-2 is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of argument-2 must be less than 10,000 and greater than 1,699.

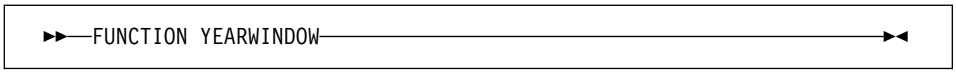
Example
Two examples of return values from the YEAR-TO-YYYY function follow:

Current Year	Argument-1 Value	Argument-2 Value	YEAR-TO-YYYY Return Value
1995	4	23	2004
1995	4	-15	1904
2008	98	23	1998
2008	98	-15	1898

YEARWINDOW Intrinsic Function
If the DATEPROC compiler option is in effect, the YEARWINDOW function returns the starting year of the century window specified by the YEARWINDOW compiler option. The returned value is an expanded date field with implicit date format YYYY.

If the NODATEPROC compiler option is in effect, the YEARWINDOW function returns 0.

The function type is integer.



The YEARWINDOW intrinsic function returns the starting year of the century window specified by the YEARWINDOW compiler option. The returned value is in expanded-year format.

Example
The following example shows a call to the intrinsic function YEARWINDOW.

```
01  STARTING-YEAR       PIC 9(4) BINARY DATE FORMAT YYYY.
    ⋮
    COMPUTE STARTING-YEAR = FUNCTION YEARWINDOW.
```

Impact on COBOL Language Elements

It is expected that the primary use of the millennium language extensions will be in changing existing COBOL programs to provide consistent date processing. However, when making such changes, you must remember that COBOL now handles date fields differently from non-dates, and COBOL statements and constructs will behave differently, or will produce a diagnostic message, when non-dates are changed to date fields.

This section outlines some of the things you need to consider when introducing date fields into a COBOL program. For full information, refer to the *IBM COBOL Language Reference* for your platform.

Considerations for Date fields

When you change a COBOL program that contains date fields with 2-digit year dates to take advantage of the millennium language extensions, the simplest change is to add DATE FORMAT clauses to these date fields to indicate that their contents should be treated as windowed dates. However, you should remember that this will affect the behavior of all statements in the program that reference these fields.

Windowed date fields carry only part of the information in themselves; the remainder of the information must come from the century window. This means that many COBOL language elements are affected by the use of windowed date fields. Expanded date fields also affect COBOL language elements, but not to the same degree as windowed dates because they hold a complete item of information.

Table 6 on page 88 lists restrictions on the use of date fields together with other COBOL language elements. Language elements can be handled in the following three ways, as outlined in the table:

Treated as non-date
: The compiler ignores the DATE FORMAT clause on the data item, and treats the contents of the field without any special date processing.

Cannot be windowed date field
: A windowed date field cannot be used in this context.

Cannot be expanded date field
: An expanded date field cannot be used in this context.

In all cases where a language element is affected, the compiler produces a diagnostic message and you should take appropriate action to correct the problem.

Table 6 (Page 1 of 2). Date Field Restrictions

Case	Language Element	Treated as non-date	Cannot be windowed date field	Cannot be expanded date field
Environment Division				
SELECT	The data-name in a USING phrase	√		
	The data-name in a PASSWORD clause	√		
	The data-name in a FILE STATUS clause	√		
	The data-name in a RECORD KEY clause		√	
	The data-name in an ALTERNATE RECORD KEY clause		√	
	The data-name in a RELATIVE KEY clause		√	
Data Division				
File Description entry	The data-name in a VARYING DEPENDING ON clause		√	
	The data-name in a LABEL RECORDS clause	√		
	Any of the data names under the LINAGE description: LINAGE, FOOTING, TOP, BOTTOM	√		
OCCURS	An identifier in the ASCENDING or DESCENDING KEY phrase		√	
	An identifier in the DEPENDING ON phrase		√	
Procedure Division				
ADD statement	As identifier-1, for a year-last date field	√[1]		
CALL statement	An identifier containing the program name.		√	
CANCEL statement	The identifier containing the program name		√	
Class condition	Any identifier	√		
COMPUTE statement	As identifier-1, for a year-last date field	√[1]		
DISPLAY statement	Any identifier containing data to display	√		
DIVIDE statement	Any field other than a receiving field		√	√
GO TO statement	An identifier in a GO TO ... DEPENDING ON statement		√	
INSPECT statement	Any identifier		√	
Intrinsic functions	An argument to any intrinsic function except UNDATE		√	

Table 6 (Page 2 of 2). Date Field Restrictions

Case	Language Element	Treated as non-date	Cannot be windowed date field	Cannot be expanded date field
INVOKE statement	An identifier containing the object or method name.		√	
MERGE statement (non-host only)[2]	A data name used as a merge key in the ASCENDING/DESCENDING KEY phrase		√	
MOVE statement	Any sending item or receiving field specified as a year-last date field.	√		
MULTIPLY statement	Any field other than a receiving field		√	√
PERFORM statement	An identifier in the TIMES phrase; an identifier in the VARYING phrase (except within the conditional expression)		√	
SEARCH statement	Serial (format 1) search: an identifier in the VARYING phrase; binary (format 2) search: any identifier or data-name (except within the conditional expression)		√	
SET statement	Any identifier		√	
SUBTRACT statement	As identifier-1, for a year-last date field	√[1]		
Sign condition	The identifier specified as *operand-1*	√		
SORT statement (non-host only)[2]	A data name used as a sort key in the ASCENDING/DESCENDING KEY phrase		√	
STRING statement	Any identifier		√	
UNSTRING statement	Any identifier		√	
WRITE statement	An identifier in the ADVANCING phrase		√	

Note:

- [1] Windowed date fields can be used with the MERGE and SORT statements on host platforms with the relevant level of DFSORT software. For details, see “Sorting and Merging” on page 23, and Chapter 5, “Using MLE with DFSORT” on page 44.
- [2] A year-last date field is allowed in an ADD, COMPUTE, or SUBTRACT statement only as identifier-1 and when the result is a non-date.

Other Data Division Restrictions

In addition to the items listed in Table 6 on page 88, there are other considerations and restrictions for the Data Division. These are listed here.

Restricted Clauses

The following clauses are not permitted with date fields:

BLANK WHEN ZERO
JUSTIFIED

SIGN clause: If the SIGN clause is used with a date field, it cannot specify the SEPARATE CHARACTER phrase.

USAGE Clause

The USAGE of a date field must be specified or assumed to be one of the following:

BINARY
COMP
COMP-3
COMPUTATIONAL
COMPUTATIONAL-3
DISPLAY
PACKED-DECIMAL

VALUE Clause

If an alphanumeric date field has a condition-name (level-88) VALUE clause, the number of characters in the value clause must exactly equal the length in bytes of the data item.

PICTURE Clause

For numeric date fields, the only picture characters permitted are S and 9. The characters P and V are not permitted.

For alphanumeric dates, any combination of the characters 9, A, and X, with at least one X, is permitted.

The PICTURE clause character-string must specify the same number of characters or digits as the DATE FORMAT clause. For example, if a PICTURE X(6) data item has a DATE FORMAT clause, the date format specification must be one of the following: `YYXXXX`, `YYYYXX`, `XXXXYY` or `XXYYYY`.

Chapter 8. Examples of MLE Usage

This chapter gives some examples of how you can use COBOL with the millennium language extensions to convert files and programs.

The examples shown here are as follows:

- "Date Formats" is an example of how the millennium language extensions constructs can be added to a program to ensure that the program continues to function after 1999.
- "Subscripting" on page 97 shows how you can use windowed year fields to create subscripts for an array.
- "File Conversion" on page 102 shows an example of a simple program that converts date fields in a file to expanded form.
- "Coordinating Century Windows" on page 103 shows how you can coordinate the century windows used by the Language Environment callable services, the COBOL intrinsic functions, and the millennium language extensions.
- "General Example" on page 105 is a sample program that shows a number of different MLE constructs. Comments in the program describe the meaning of the COBOL code.

Date Formats

The program in Figure 3 is an example of a pre-MLE program that manipulates date fields with 2-digit years. The program reads a VSAM file of employee records, and checks that the current employees still have security clearance. For those who don't, the security expiry date is extended by one year.

Because this program uses only 2-digit years in its date calculations and comparisons, it will not work correctly after 1999.

```
CBL QUOTE SOURCE LIST XREF MAP
IDENTIFICATION DIVISION.
PROGRAM-ID.   EMPSEC.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.
```

Figure 3 (Part 1 of 4). Using Date Fields with 2-digit Years

Examples of MLE Usage

```
       FILE-CONTROL.

           SELECT EMPLOYEE-MASTER-FILE
                  ASSIGN TO EMPMAST
                  ORGANIZATION IS INDEXED
                  ACCESS IS SEQUENTIAL
                  RECORD KEY IS EMPLOYEE-ID
                  FILE STATUS IS EMP-FILE-STATUS.

           SELECT PRINT-EDIT-REPORT
                  ASSIGN TO SYSPRINT
                  FILE STATUS IS PRT-FILE-STATUS.

       DATA DIVISION.

       FILE SECTION.

       FD  EMPLOYEE-MASTER-FILE
           RECORD CONTAINS 200 CHARACTERS.

       01  EMPLOYEE-MASTER-RECORD.
      *        ** Key field
           03  EMPLOYEE-ID                 PIC X(6).
           03  EMPLOYEE-DEPT-CODE          PIC X(4).
           03  EMPLOYEE-STATUS             PIC X.
               88  EMPLOYEE-CURRENT                VALUE "C".
               88  EMPLOYEE-TERMINATED             VALUE "T".
           03  EMPLOYEE-NAME               PIC X(30).
           03  EMPLOYEE-ADDR-1             PIC X(30).
           03  EMPLOYEE-ADDR-2             PIC X(30).
           03  EMPLOYEE-ADDR-3             PIC X(30).
           03  EMPLOYEE-ZIP-CODE           PIC X(5).
      *        ** All date formats (yymmdd)
           03  EMPLOYEE-DATE-JOINED        PIC 9(6).
           03  EMPLOYEE-DATE-TERMINATED    PIC 9(6).
           03  EMPLOYEE-DATE-MAINTAINED    PIC 9(6).
           03  EMPLOYEE-BIRTH-DATE         PIC 9(6).
           03  EMPLOYEE-SECURITY-EXP       PIC 9(6).
           03  FILLER                      PIC X(35).

       FD  PRINT-EDIT-REPORT
           RECORDING MODE IS F
           RECORD CONTAINS 132 CHARACTERS
           LABEL RECORDS ARE OMITTED.
       01  PRINT-RECORD                    PIC X(132).

       WORKING-STORAGE SECTION.

       01  FILE-STATUS-FIELDS.
           03  EMP-FILE-STATUS             PIC XX.
           03  PRT-FILE-STATUS             PIC XX.
```

Figure 3 (Part 2 of 4). Using Date Fields with 2-digit Years

```
 01  WORK-DATE-YYMMDD               PIC 9(6).
 01  WORK-CURRENT-EXP               PIC 9(6).
 01  WORK-NEW-EXP.
     03  WORK-NEW-YEAR              PIC 99.
     03  WORK-NEW-MONTH-DAY         PIC 9(4).

*    **  Print line data goes here

 PROCEDURE DIVISION.

     PERFORM 100-INITIALIZE THRU 100-EXIT.

     PERFORM 200-MAIN-PROCESS THRU 200-EXIT.

     STOP RUN.

 100-INITIALIZE.

*    **  Get today's date

     ACCEPT WORK-DATE-YYMMDD FROM DATE.

*    **  Open files

     OPEN I-O EMPLOYEE-MASTER-FILE.
     OPEN OUTPUT PRINT-EDIT-REPORT.

 100-EXIT.
     EXIT.

 200-MAIN-PROCESS.

     READ EMPLOYEE-MASTER-FILE
         AT END
         GO TO 200-EXIT
     END-READ.

     IF  EMPLOYEE-CURRENT THEN
         IF  EMPLOYEE-SECURITY-EXP <= WORK-DATE-YYMMDD THEN
             MOVE EMPLOYEE-SECURITY-EXP TO WORK-CURRENT-EXP
                                           WORK-NEW-EXP
             ADD 1 TO  WORK-NEW-YEAR
             IF  WORK-NEW-MONTH-DAY = 0229 THEN
                 MOVE 0228 TO WORK-NEW-MONTH-DAY
             END-IF

             PERFORM 300-PRINT-REPORT THRU 300-EXIT
             PERFORM 400-UPDATE-FILE  THRU 400-EXIT
         END-IF
     END-IF.

     GO TO 200-MAIN-PROCESS.

 200-EXIT.
     EXIT.
```

Figure 3 (Part 3 of 4). Using Date Fields with 2-digit Years

Examples of MLE Usage

```
       300-PRINT-REPORT.

      *    **  Code to print report

       300-EXIT.
           EXIT.

       400-UPDATE-FILE.

           MOVE WORK-NEW-EXP TO EMPLOYEE-SECURITY-EXP.
           REWRITE EMPLOYEE-MASTER-RECORD.

       400-EXIT.
           EXIT.
```

Figure 3 (Part 4 of 4). Using Date Fields with 2-digit Years

Figure 4 shows the same program with changes to take advantage of the millennium language extensions features. This ensures that the program will continue to work correctly after 1999.

```
       CBL QUOTE SOURCE LIST XREF MAP DATEPROC(FLAG) YEARWINDOW(-90) 1
       IDENTIFICATION DIVISION.
       PROGRAM-ID.   EMPSEC.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.

       FILE-CONTROL.

           SELECT EMPLOYEE-MASTER-FILE
                  ASSIGN TO EMPMAST
                  ORGANIZATION IS INDEXED
                  ACCESS IS SEQUENTIAL
                  RECORD KEY IS EMPLOYEE-ID
                  FILE STATUS IS EMP-FILE-STATUS.

           SELECT PRINT-EDIT-REPORT
                  ASSIGN TO SYSPRINT
                  FILE STATUS IS PRT-FILE-STATUS.

       DATA DIVISION.

       FILE SECTION.

       FD  EMPLOYEE-MASTER-FILE
           RECORD CONTAINS 200 CHARACTERS.
```

Figure 4 (Part 1 of 3). Using DATE FORMAT to Process 2-digit Years

```
 01  EMPLOYEE-MASTER-RECORD.
*        ** Key field
     03  EMPLOYEE-ID                 PIC X(6).
     03  EMPLOYEE-DEPT-CODE          PIC X(4).
     03  EMPLOYEE-STATUS             PIC X.
         88  EMPLOYEE-CURRENT                VALUE "C".
         88  EMPLOYEE-TERMINATED             VALUE "T".
     03  EMPLOYEE-NAME               PIC X(30).
     03  EMPLOYEE-ADDR-1             PIC X(30).
     03  EMPLOYEE-ADDR-2             PIC X(30).
     03  EMPLOYEE-ADDR-3             PIC X(30).
     03  EMPLOYEE-ZIP-CODE           PIC X(5).
*        ** All date formats (yymmdd)
     03  EMPLOYEE-DATE-JOINED        PIC 9(6) DATE FORMAT YYXXXX. 2
     03  EMPLOYEE-DATE-TERMINATED    PIC 9(6) DATE FORMAT YYXXXX.
     03  EMPLOYEE-DATE-MAINTAINED    PIC 9(6) DATE FORMAT YYXXXX.
     03  EMPLOYEE-BIRTH-DATE         PIC 9(6) DATE FORMAT YYXXXX.
     03  EMPLOYEE-SECURITY-EXP       PIC 9(6) DATE FORMAT YYXXXX.
     03  FILLER                      PIC X(35).

 FD  PRINT-EDIT-REPORT
     RECORDING MODE IS F
     RECORD CONTAINS 132 CHARACTERS
     LABEL RECORDS ARE OMITTED.
 01  PRINT-RECORD                    PIC X(132).

 WORKING-STORAGE SECTION.

 01  FILE-STATUS-FIELDS.
     03  EMP-FILE-STATUS             PIC XX.
     03  PRT-FILE-STATUS             PIC XX.

 01  WORK-DATE-YYMMDD                PIC 9(6) DATE FORMAT YYXXXX. 3
 01  WORK-CURRENT-EXP                PIC 9(6) DATE FORMAT YYXXXX.
 01  WORK-NEW-EXP                            DATE FORMAT YYXXXX.
     03  WORK-NEW-YEAR               PIC 99   DATE FORMAT YY.
     03  WORK-NEW-MONTH-DAY          PIC 9(4).

*    **  Print line data goes here

 PROCEDURE DIVISION.

     PERFORM 100-INITIALIZE THRU 100-EXIT.

     PERFORM 200-MAIN-PROCESS THRU 200-EXIT.

     STOP RUN.
```

Figure 4 (Part 2 of 3). Using DATE FORMAT to Process 2-digit Years

Examples of MLE Usage

```
 100-INITIALIZE.

*    **  Retrieve today's date

     ACCEPT WORK-DATE-YYMMDD FROM DATE.        4

*    **  Open files

     OPEN I-O EMPLOYEE-MASTER-FILE.
     OPEN OUTPUT PRINT-EDIT-REPORT.

 100-EXIT.
     EXIT.

 200-MAIN-PROCESS.

     READ EMPLOYEE-MASTER-FILE
         AT END
         GO TO 200-EXIT
     END-READ.

     IF  EMPLOYEE-CURRENT THEN
         IF  EMPLOYEE-SECURITY-EXP <= WORK-DATE-YYMMDD THEN   5
             MOVE EMPLOYEE-SECURITY-EXP TO WORK-CURRENT-EXP
                                           WORK-NEW-EXP
             ADD 1 TO  WORK-NEW-YEAR     6
             IF  WORK-NEW-MONTH-DAY = 0229 THEN
                 MOVE 0228 TO WORK-NEW-MONTH-DAY
             END-IF

             PERFORM 300-PRINT-REPORT THRU 300-EXIT
             PERFORM 400-UPDATE-FILE  THRU 400-EXIT
         END-IF
     END-IF.

     GO TO 200-MAIN-PROCESS.

 200-EXIT.
     EXIT.

 300-PRINT-REPORT.

*    **  Code to print report

 300-EXIT.
     EXIT.

 400-UPDATE-FILE.

     MOVE WORK-NEW-EXP TO EMPLOYEE-SECURITY-EXP.
     REWRITE EMPLOYEE-MASTER-RECORD.

 400-EXIT.
     EXIT.
```

Figure 4 (Part 3 of 3). Using DATE FORMAT to Process 2-digit Years

The following notes apply to Figure 4 on page 94.

1 The DATEPROC compiler option enables the millennium language extensions. The YEARWINDOW compiler option specifies the century window to be used for this compile unit.

2 The date fields in the VSAM record have had the DATE FORMAT clause added so that COBOL will recognize them as windowed date fields.

3 The work date fields have also had the DATE FORMAT clause added.

4 The ACCEPT statement is not changed; the field into which the system date is being received is now a windowed date field, so COBOL handles the year component consistently.

5 Because both fields in this comparison are are now windowed date fields, COBOL ensures that the comparison is consistent. For example, if EMPLOYEE-SECURITY-EXP contains 990827 and WORK-DATE-YYMMDD contains 000412, EMPLOYEE-SECURITY-EXP will actually be *less than* WORK-DATE-YYMMDD (assuming an appropriate century window). If they were not date fields, the result of the comparison would be incorrect.

6 Because WORK-NEW-YEAR is a windowed date field, COBOL ensures consistency in this ADD statement. If, for example, WORK-NEW-YEAR contains 99, this is interpreted as 1999 by virtue of the century window, so adding 1 gives a result of 2000. This result is stored as 00 in WORK-NEW-YEAR, but it is recognized as 2000 because of the century window setting.

Note that there are no changes to the Procedure Division in this program. Changing the fields in the Data Division is sufficient to ensure that the program will continue to work properly after 1999. However, it is still necessary to review the Procedure Division to verify that the program will behave correctly with the new windowed date fields.

Subscripting

In the sample program in Figure 5 on page 98, there is a table of 20 elements representing sales amounts for the past 20 years. The subscript used to index the table is based on the year in which the program is running, such that:

- Element 1 represents the sales amount for 20 years ago
- Element 2 is for 19 years ago
- Element 19 is for 2 years ago
- Element 20 is for last year

This creates a sliding 20-year window for the purpose of the sales report. However, as you can see, all year fields in the program use two digits only, so the program will only work correctly up to 1999.

Examples of MLE Usage

```
CBL  LIB,QUOTE
      *=============================================================*
      * SALESTAB - Read a file of sales records and accumulate sales   *
      *            totals for the last 20 years.                    *
      *=============================================================*
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  SALESTAB.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT SALES-FILE
                  ASSIGN TO SALES
                  FILE STATUS IS SALES-FILE-STATUS.

       DATA DIVISION.

       FILE SECTION.

       FD  SALES-FILE
           RECORDING MODE IS F.
       01  SALES-RECORD.
           03  DEPARTMENT      PIC X(10).
           03  SALESMAN        PIC X(8).
           03  SALE-DATE.
               05  SALE-YEAR   PIC 99.
               05  SALE-MONTH  PIC 99.
               05  SALE-DAY    PIC 99.
           03  SALE-ITEM       PIC X(8).
           03  SALE-AMOUNT     PIC S9(5)V99.

       WORKING-STORAGE SECTION.

       01  SALES-FILE-STATUS   PIC 99.

      *-------------------------------------------------------------*
      *    Table of 20 sales amount totals.                         *
      *-------------------------------------------------------------*
       01  SALES-TABLE.
           03  SALES-AMOUNT    PIC S9(9)V99 OCCURS 20 TIMES.

      *-------------------------------------------------------------*
      *    Subscript into the sales amount table.                   *
      *-------------------------------------------------------------*
       01  SALES-INDEX         PIC 9(4) COMP.

       01  FIRST-YEAR          PIC 99.
       01  LAST-YEAR           PIC 99.

       01  TODAYS-DATE.
           03  THIS-YEAR       PIC 99.
           03                  PIC X(4).
```

Figure 5 (Part 1 of 2). Using the Year as a Subscript

```
 PROCEDURE DIVISION.

     INITIALIZE SALES-TABLE.
     ACCEPT TODAYS-DATE FROM DATE.
     SUBTRACT 20 FROM THIS-YEAR GIVING FIRST-YEAR.
     SUBTRACT 1  FROM THIS-YEAR GIVING LAST-YEAR.

     OPEN INPUT SALES-FILE.

 READ-RECORD.
*-----------------------------------------------------------------*
*    Read each record and check that the year falls within our   *
*    20-year time frame.                                         *
*-----------------------------------------------------------------*
     READ SALES-FILE
          AT END GO TO CLOSE-FILE.
     IF SALE-YEAR LESS THAN FIRST-YEAR OR
        SALE-YEAR GREATER THAN LAST-YEAR THEN
        GO TO READ-RECORD.

*-----------------------------------------------------------------*
*    Calculate the index into the table and add the amount to    *
*    that table entry.                                           *
*-----------------------------------------------------------------*
     COMPUTE SALES-INDEX = SALE-YEAR - FIRST-YEAR + 1.
     ADD SALE-AMOUNT TO SALES-AMOUNT(SALES-INDEX).

*-----------------------------------------------------------------*
*    Get the next record.                                        *
*-----------------------------------------------------------------*
     GO TO READ-RECORD.

 CLOSE-FILE.
     CLOSE SALES-FILE.

*-----------------------------------------------------------------*
*    Process the information in the table.                       *
*-----------------------------------------------------------------*
     CALL "PROCTAB" USING SALES-TABLE.

*-----------------------------------------------------------------*
*    End of SALESTAB program.                                    *
*-----------------------------------------------------------------*
     EXIT PROGRAM.

 END PROGRAM SALESTAB.
```

Figure 5 (Part 2 of 2). Using the Year as a Subscript

The same program is shown in Figure 6 on page 100 with changes to allow for the 2-digit year in the input file to be windowed. These changes mean that the sliding 20-year window of the sales report simply becomes part of the sliding 100-year window of the COBOL compile unit.

Examples of MLE Usage

```
CBL  LIB,QUOTE,DATEPROC(FLAG),YEARWINDOW(-90)          1
      *===============================================================*
      * SALESTAB - Read a file of sales records and accumulate sales  *
      *            totals for the last 20 years.                      *
      *===============================================================*
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  SALESTAB.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT SALES-FILE
                  ASSIGN TO SALES
                  FILE STATUS IS SALES-FILE-STATUS.

       DATA DIVISION.

       FILE SECTION.

       FD  SALES-FILE
           RECORDING MODE IS F.
       01  SALES-RECORD.
           03  DEPARTMENT      PIC X(10).
           03  SALESMAN        PIC X(8).
           03  SALE-DATE.
               05  SALE-YEAR   PIC 99 DATE FORMAT YY.  2
               05  SALE-MONTH  PIC 99.
               05  SALE-DAY    PIC 99.
           03  SALE-ITEM       PIC X(8).
           03  SALE-AMOUNT     PIC S9(5)V99.

       WORKING-STORAGE SECTION.

       01  SALES-FILE-STATUS   PIC 99.

      *---------------------------------------------------------------*
      *    Table of 20 sales amount totals.                           *
      *---------------------------------------------------------------*
       01  SALES-TABLE.
           03  SALES-AMOUNT    PIC S9(9)V99 OCCURS 20 TIMES.

      *---------------------------------------------------------------*
      *    Subscript into the sales amount table.                     *
      *---------------------------------------------------------------*
       01  SALES-INDEX         PIC 9(4) COMP.

       01  FIRST-YEAR          PIC 9(4) DATE FORMAT YYYY.    3
       01  LAST-YEAR           PIC 9(4) DATE FORMAT YYYY.

       01  TODAYS-DATE                  DATE FORMAT YYYYXXXX.
           03  THIS-YEAR       PIC 9(4) DATE FORMAT YYYY.    4
           03                  PIC X(4).

       PROCEDURE DIVISION.

           INITIALIZE SALES-TABLE.
           ACCEPT TODAYS-DATE FROM DATE YYYYMMDD.            5
           SUBTRACT 20 FROM THIS-YEAR GIVING FIRST-YEAR.
           SUBTRACT 1  FROM THIS-YEAR GIVING LAST-YEAR.

           OPEN INPUT SALES-FILE.
```

Figure 6 (Part 1 of 2). Using the Windowed Year as a Subscript

```
 READ-RECORD.
*-----------------------------------------------------------------*
*    Read each record and check that the year falls within our   *
*    20-year time frame.                                         *
*-----------------------------------------------------------------*
     READ SALES-FILE
          AT END GO TO CLOSE-FILE.
     IF SALE-YEAR LESS THAN FIRST-YEAR OR                    6
        SALE-YEAR GREATER THAN LAST-YEAR THEN
        GO TO READ-RECORD.

*-----------------------------------------------------------------*
*    Calculate the index into the table and add the amount to    *
*    that table entry.                                           *
*-----------------------------------------------------------------*
     COMPUTE SALES-INDEX = SALE-YEAR - FIRST-YEAR + 1.       7
     ADD SALE-AMOUNT TO SALES-AMOUNT(SALES-INDEX).

*-----------------------------------------------------------------*
*    Get the next record.                                        *
*-----------------------------------------------------------------*
     GO TO READ-RECORD.

 CLOSE-FILE.
     CLOSE SALES-FILE.

*-----------------------------------------------------------------*
*    Process the information in the table.                       *
*-----------------------------------------------------------------*
     CALL "PROCTAB" USING SALES-TABLE.

*-----------------------------------------------------------------*
*    End of SALESTAB program.                                    *
*-----------------------------------------------------------------*
     EXIT PROGRAM.

 END PROGRAM SALESTAB.
```

Figure 6 (Part 2 of 2). Using the Windowed Year as a Subscript

The following notes apply to Figure 6 on page 100.

1 The DATEPROC compiler option enables the millennium language extensions. The YEARWINDOW compiler option specifies the century window to be used for this compile unit.

2 The SALE-YEAR field has a DATE FORMAT clause added so it will be recognized as a windowed year field.

3 The fields FIRST-YEAR and LAST-YEAR have been expanded from two digits to four.

4 The field TODAYS-DATE has been expanded to accommodate a 4-digit year, and defined as a expanded date field.

5 The ACCEPT statement now includes the YYYYMMDD construct. This causes the system date to be retrieved in expanded form. The SUBTRACT statements are unchanged; the numbers are subtracted from the 4-digit current year to give the 4-digit starting and ending years of the 20-year period.

6 The IF statement compares SALE-YEAR (a windowed year field) to FIRST-YEAR and LAST-YEAR (expanded year fields). COBOL applies the century window to SALE-YEAR to make the comparison consistent.

7 SALES-INDEX is the subscript used to address the table. Its value (between 1 and 20) is calculated as the difference between SALE-YEAR and FIRST-YEAR. As these are both date fields, the calculation is consistent across century boundaries.

File Conversion

The program in Figure 7 is a simple program that copies from one file to another while converting the date fields in the file to expanded form. The record length of the output file is larger than that of the input file because the dates are expanded, so a program of this nature would normally be run once only as part of a conversion process.

```
CBL  LIB,QUOTE,NOOPT,DATEPROC(FLAG),YEARWINDOW(-80)
      ***********************************************
      ** CONVERT - Read a file, convert the date    **
      **           fields to expanded form, write   **
      **           the expanded records to a new    **
      **           file.                            **
      ***********************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CONVERT.

       ENVIRONMENT DIVISION.

       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INPUT-FILE
                  ASSIGN TO INFILE
                  FILE STATUS IS INPUT-FILE-STATUS.

           SELECT OUTPUT-FILE
                  ASSIGN TO OUTFILE
                  FILE STATUS IS OUTPUT-FILE-STATUS.

       DATA DIVISION.
       FILE SECTION.
       FD  INPUT-FILE
           RECORDING MODE IS F.
       01  INPUT-RECORD.
           03  CUST-NAME.
               05  FIRST-NAME  PIC X(10).
               05  LAST-NAME   PIC X(15).
           03  ACCOUNT-NUM     PIC 9(8).
           03  DUE-DATE        PIC X(6) DATE FORMAT YYXXXX.       1
           03  REMINDER-DATE   PIC X(6) DATE FORMAT YYXXXX.
           03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.
```

Figure 7 (Part 1 of 2). Expanding File Dates

```
FD  OUTPUT-FILE
    RECORDING MODE IS F.
01  OUTPUT-RECORD.
    03  CUST-NAME.
        05  FIRST-NAME  PIC X(10).
        05  LAST-NAME   PIC X(15).
    03  ACCOUNT-NUM     PIC 9(8).
    03  DUE-DATE        PIC X(8) DATE FORMAT YYYYXXXX.   2
    03  REMINDER-DATE   PIC X(8) DATE FORMAT YYYYXXXX.
    03  DUE-AMOUNT      PIC S9(5)V99 COMP-3.

WORKING-STORAGE SECTION.

01  INPUT-FILE-STATUS   PIC 99.
01  OUTPUT-FILE-STATUS  PIC 99.

PROCEDURE DIVISION.

    OPEN INPUT INPUT-FILE.
    OPEN OUTPUT OUTPUT-FILE.

READ-RECORD.
    READ INPUT-FILE
         AT END GO TO CLOSE-FILES.
    MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.        3
    WRITE OUTPUT-RECORD.

    GO TO READ-RECORD.

CLOSE-FILES.
    CLOSE INPUT-FILE.
    CLOSE OUTPUT-FILE.

    EXIT PROGRAM.

END PROGRAM CONVERT.
```

Figure 7 (Part 2 of 2). Expanding File Dates

The following notes apply to Figure 7 on page 102.

1 The fields DUE-DATE and REMINDER-DATE in the input record are both Gregorian dates with 2-digit year components. They have been defined with a DATE FORMAT clause in this program so that the compiler will recognize them as windowed date fields.

2 The output record contains the same two fields in expanded date format. They have been defined with a DATE FORMAT clause so that the compiler will treat them as 4-digit year date fields.

3 The MOVE CORRESPONDING statement moves each item in INPUT-RECORD individually to its matching item in OUTPUT-RECORD. When the two windowed date fields are moved to the corresponding expanded date fields, the compiler will expand the year values using the current century window.

Coordinating Century Windows

Language Environment provides several callable services to perform date and time functions such as formatting and conversion. Some of these services can act on a date

containing a 2-digit year, and in these cases Language Environment applies its own century window to determine the expanded year. By default, this century window starts 80 years prior to the current year, and can only be changed by the CEESCEN callable service. By contrast, the COBOL century window is determined by the YEARWINDOW compiler option, and cannot be changed during the execution of the program.

COBOL also provides intrinsic functions to perform date expansion services, and these functions need to know the century window to use for the expansion. These functions are:

DATE-TO-YYYYMMDD
DAY-TO-YYYYDDD
YEAR-TO-YYYY

The second argument to these functions defines the century window to be applied to the date in the first argument. This century window is specified as a number which, when added to the current year at the time of execution, gives the ending year of the window. For full syntax details of these intrinsic functions, see “Intrinsic Functions” on page 80, and the *IBM COBOL Language Reference* for your platform.

Figure 8 shows how you can interrogate the COBOL century window using the YEARWINDOW intrinsic function, and use the result to:

- Change the Language Environment century window, and
- Define a field that can be used in calls to the COBOL intrinsic functions to define their century window.

This will ensure that all century windows are equal, which in turn will ensure consistency between COBOL date field operations, Language Environment callable services, and COBOL intrinsic functions.

```
 01  LE-FEEDBACK           PIC X(12).
 01  LE-START-POINT        PIC 9(9) BINARY.
 01  COBOL-START-YEAR      PIC 9(4) DATE FORMAT YYYY.
 01  COBOL-INTRINSIC-ARG   PIC 9(4).
 01  CURRENT-YEAR          PIC 9(4) DATE FORMAT YYYY.
     ⋮
* Get the current year
     MOVE FUNCTION CURRENT-DATE(1:4) TO CURRENT-YEAR.

* Get the starting year of the COBOL century window
     COMPUTE COBOL-START-YEAR = FUNCTION YEARWINDOW.

* Calculate the Language Environment starting year
     SUBTRACT COBOL-START-YEAR FROM CURRENT-YEAR
              GIVING LE-START-POINT.

* Tell LE about it
     CALL "CEESCEN" USING LE-START-POINT, LE-FEEDBACK.

* Calculate the argument to use in COBOL intrinsic functions
     COMPUTE COBOL-INTRINSIC-ARG = COBOL-START-YEAR
                                 + 99
                                 - CURRENT-YEAR.
```

Figure 8. Matching the Century Windows

General Example

Figure 9 shows a sample COBOL program that contains a number of different uses of MLE constructs, and demonstrates a number of different methods of implementing date processing. The program is annotated with comments that describe the meaning of the code.

```
       PROCESS FLAG(I,I) DATEPROC YEARWINDOW(1950)
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SOLUTION.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT CUSTOMER-FILE ASSIGN CUSTFILE.
       DATA DIVISION.
       FILE SECTION.
       SD CUSTOMER-FILE.
      ****************************************************************
      *  Date fields used for multiple purposes
      *  1) MMDDYY stored in YYMMDD
      ****************************************************************
       01  CUST-RECORD.
         02  CUST-SORT-KEY.
           05  BLAH-BLAH1    PIC 9(9) BINARY.
           05  BLAH-BLAH2    PIC X(30).
      *     Add redefinition or add new data item
new          05  CUST-DATE2.
old   *    05  CUST-DATE     PIC 9(6).
new          07  CUST-DATE   PIC 9(6) DATE FORMAT YYXXXX.
         02  KEY-SAVE        PIC X(40).
       WORKING-STORAGE SECTION.
         77  SQL-FROM        PIC X(80).
         77  WIN-OFFSET      PIC 9(2).
         77  NON-YEAR        PIC 9(3).
         77  DATE-SWITCH     PIC X.
         77  NON-DATE        PIC 9(6).
         77  DURATION        PIC 9(6).
         01.
           02  TAB1          PIC 9(6) OCCURS 100 TIMES.
         77  EXP-CUST-DATE   PIC 9(8)  DATE FORMAT YYYYXXXX.
         01  CURR-DATE                  DATE FORMAT YYYYXXXX.
           05  CURR-YEAR     PIC 9(4)  DATE FORMAT YYYY.
           05  CURR-MONTH    PIC 99.
           05  CURR-DAY      PIC 99.
         01  END-DATE                   DATE FORMAT YYXXXX.
           05  END-YEAR      PIC 99    DATE FORMAT YY.
           05  END-MONTH     PIC 99.
           05  END-DAY       PIC 99.
         01  EXP-DATE        PIC X(8)  DATE FORMAT YYYYXXXX.
         77 YYYY             PIC 9(4)  DATE FORMAT YYYY.
         77 ANSI-PIVOT       PIC 9(4)  DATE FORMAT YYYY.
```

Figure 9 (Part 1 of 5). General Sample Program

Examples of MLE Usage

```
   01 MORE-DATE-FORMATS.
*****************************************************************
*  More date formats.  You can use YYXX as Year and
*  month or as year and week, but don't mix them!
*****************************************************************
      02 YYMMDD1               DATE FORMAT YYXXXX.
         03 YYMM               DATE FORMAT YYXX.
            05 YY1 PIC 99      DATE FORMAT YY.
            05 MM1 PIC 99.
         03 DD1    PIC 99.
      02 YYWW      PIC 9(4)    DATE FORMAT YYXX.
*****************************************************************
*  More date formats.  To handle MMDDYY variation for
*  ordering make sure year is described separately so
*  that DATE FORMAT YY can be added. If used in < or >
*  in existing code, then this has to be in place already.
*  If not used in < or > then can leave as non-date.
*****************************************************************
      02 MMDDYY    PIC 9(6).
      02 NEW-DEF REDEFINES MMDDYY.
         03 DD2    PIC 99.
         03 MM2    PIC 99.
         03 YY2    PIC 99      DATE FORMAT YY.
*****************************************************************
*  Century field at different level from year.
*****************************************************************
      02 CCYYMMDD3             DATE FORMAT YYYYXXXX.
         03 CC3    PIC 99.
         03 YYMMDD3            DATE FORMAT YYXXXX.
            05 YY3 PIC 99      DATE FORMAT YY.
            05 MM3 PIC 99.
            05 DD3 PIC 99.

 PROCEDURE DIVISION.
*****************************************************************
* How to implement multiple windows
* Dates with non-MLE window must be defined as non-dates
* Use Intrinsic Functions to expand the dates in-line, result
* will be determined using the default sliding century window
* (current-year - 49 to current-year + 50).
*****************************************************************
*    IF CUST-DATE > NON-DATE THEN
     IF CUST-DATE > FUNCTION DATE-TO-YYYYMMDD( NON-DATE ) THEN
       CONTINUE.
*****************************************************************
* Use Intrinsic Functions to expand a date in-line, but this
* time use 1920 as the base year instead of the default.
* WIN-OFFSET can be used over and over to specify a pivot of
* 1920, which in ANSI terms is specified as ending in 2019.
*****************************************************************
     ACCEPT CURR-DATE FROM DATE YYYYMMDD.
     COMPUTE ANSI-PIVOT = 2019.
     COMPUTE WIN-OFFSET = CURR-YEAR - ANSI-PIVOT
     IF CUST-DATE > FUNCTION
                    DATE-TO-YYYYMMDD( NON-DATE WIN-OFFSET) THEN
       CONTINUE.
```

Figure 9 (Part 2 of 5). General Sample Program

```
*****************************************************************
*  Special values, trigger values
*  If year 99 means trigger and not 1999, then
*  use the whole date field. 999999 is > 991231
*****************************************************************
*    IF END-YEAR < 99 THEN
     IF FUNCTION UNDATE (END-DATE) < 999999 THEN

*****************************************************************
*  Windowed SORT for DFSORT.  If date is embedded
*  in KEY, then need descriptions of sub fields of
*  the KEY and list them separately:
*****************************************************************

*    SORT CUSTOMER-FILE ASCENDING KEY CUST-SORT-KEY
     SORT CUSTOMER-FILE ASCENDING KEY BLAH-BLAH1
                                      BLAH-BLAH2
                                      CUST-DATE
                        INPUT  PROCEDURE INPROC
                        OUTPUT PROCEDURE OUTPROC.

*****************************************************************
*  More date formats.  You can use YYXX as Year and
*  month or year and week, but don't mix them!
*  Here is an example of how NOT to do it.
*****************************************************************

     IF YYMM > YYWW THEN
       DISPLAY "Months might never be greater than weeks!"
     END-IF

*****************************************************************
*  More date formats.  To handle MMDDYY variation for
*  ordering make sure year is described separately so that
*  DATE FORMAT YY can be added.  These moves move
*  YYMMDD into MMDDYY.
*****************************************************************

     MOVE YY1 TO YY2.
     MOVE MM1 TO MM2.
     MOVE DD1 TO DD2.

*****************************************************************
*  More date formats.  To handle MMDDYY variation for
*  ordering make sure year is described separately so that
*  DATE FORMAT YY can be added.  Once the YEAR is MLE'd
*  then existing ordering will work in cases such as:
*****************************************************************
     IF YY1 > YY2 OR
        (YY1 = YY2 AND
         (MM1 > MM2 OR (MM1 = MM2 AND DD1 > DD2))) THEN
       DISPLAY 'MMDDYY handled with MLE!'
     END-IF
```

Figure 9 (Part 3 of 5). General Sample Program

Examples of MLE Usage

```
*****************************************************************
*  Reference-modified dates.  How to use reference
*  modification after a date field has been identified:
*****************************************************************

*    MOVE YYMMDD1(1:2) TO MMDDYY(5:2).
*    MOVE YYMMDD1(3:2) TO MMDDYY(3:2).
*    MOVE YYMMDD1(5:2) TO MMDDYY(1:2).
     MOVE FUNCTION UNDATE (YYMMDD1)(1:2) TO MMDDYY(5:2).
     MOVE FUNCTION UNDATE (YYMMDD1)(3:2) TO MMDDYY(3:2).
     MOVE FUNCTION UNDATE (YYMMDD1)(5:2) TO MMDDYY(1:2).

*****************************************************************
*  Mixing MLE date processing with existing windowing
*  logic, both in-line and subroutine methods.
*****************************************************************
* BEFORE:
*    ACCEPT YYMMDD3 FROM DATE.
*    IF YY3 > 50 THEN
*      MOVE 19 TO CC3
*    ELSE
*      MOVE 20 TO CC3.

* AFTER1: (only 1 line!)
     ACCEPT CCYYMMDD3 FROM DATE YYYYMMDD.

* AFTER2:
     ACCEPT YYMMDD3 FROM DATE.
     IF FUNCTION UNDATE(YY3) > 50 THEN
       MOVE 19 TO CC3
     ELSE
       MOVE 20 TO CC3.

*****************************************************************
*  Embedded 2-digit year dates.  When 2-digit years
*  are 'hidden' in a string, they must be described
*  separately.
*****************************************************************

*    IF CUST-SORT-KEY > KEY-SAVE THEN
     IF CUST-SORT-KEY(1:34) > KEY-SAVE(1:34) AND
        CUST-DATE > FUNCTION DATEVAL(KEY-SAVE(35:6),
                                     'YYXXXX' )  THEN
       DISPLAY 'Embedded dates!'.

*****************************************************************
*  Date fields used for multiple purposes
*  1) MMDDYY stored in YYMMDD
*     Add redefinition or add new data item
*****************************************************************
     IF DATE-SWITCH = 'Y' THEN
       MOVE YYMMDD1 TO CUST-DATE
     ELSE
*      MOVE MMDDYY TO CUST-DATE
       MOVE MMDDYY TO CUST-DATE2
     END-IF
```

Figure 9 (Part 4 of 5). General Sample Program

```
*****************************************************************
*  Date fields used for multiple purposes
*  2) Duration stored in YY
*     Add redefinition or new data item
*****************************************************************
*    COMPUTE YY1 = YY2 - YY3
     COMPUTE DURATION = YY2 - YY3

*****************************************************************
*  Using date fields in PERFORM VARYING.  If using years, then
*  any solution requires understanding of what index 00 means.
*****************************************************************
*    PERFORM VARYING YY1 FROM 1 BY 1 UNTIL YY1 = 99
*      DISPLAY TAB1(YY1)
*    END-PERFORM
*****************************************************************
*  One solution, redefine table as having 200 entries.
*  Another solution, use expanded years, subscript using - 1900
*****************************************************************
     PERFORM VARYING NON-YEAR FROM 1 BY 1 UNTIL NON-YEAR = 199
       DISPLAY TAB1(NON-YEAR)
     END-PERFORM
*****************************************************************
*  Another solution, use expanded years, subscript using - 1950
*  and just move the range of 100 table items from 1900-1999 to
*  1950-2049
*****************************************************************
     MOVE YY1 TO YYYY.
     PERFORM VARYING YYYY FROM 1950 BY 1 UNTIL YYYY = 2049
       DISPLAY TAB1(YYYY - 1950)
     END-PERFORM.

 INPROC.
     DISPLAY "INPUT".
 OUTPROC.
     DISPLAY "OUTPUT".

     GOBACK.
 END PROGRAM SOLUTION.
```

Figure 9 (Part 5 of 5). General Sample Program

Appendix A. MLE Messages

This appendix lists the MLE diagnostic messages and guidance as to when and how to use them.

Using MLE Messages

The easiest way to use the MLE messages is to compile with a FLAG option setting that will imbed the messages in the source listing following the line to which the messages refer. You can choose whether you want to see all MLE messages or just select severities.

To see all MLE messages, specify the FLAG(I,I) and dateproc (flag) compiler options. Initially, you might want to see all of the messages to understand how MLE is processing the date fields in your program. For example, if you want to do a static analysis of the date usage in a program by using the compile listing, then you will need to use FLAG (I,I).

However, it's recommended that you specify FLAG(W,W) for MLE-specific compiles. You must resolve all Severe-level (S-level) error messages, and you should resolve all of the Error-level (E-level) messages as well. For the Warning-level (W-level) messages, if you agree with the assumptions that the compiler has made, you can ignore the message and your code will compile correctly. If you disagree with the assumption, you'll need to change the code. In some cases, the W-level message might be acceptable, but you might want to change the code to get a return code = 0 compile.

This section should help you determine what action to take when you change your code in response to an MLE diagnostic message. Look up the number of the message and find the programmer response to see what suggestions are provided as possible responses to that message.

MLE Messages Description

Messages for COBOL contain a message prefix, compiler phase indicator, message number, severity code, and descriptive text. The message prefix is always IGY, followed by the phase indicator (some of the indicators are DS for Data Scan, GR for Group Data analysis phase, PS for Procedure Scan and PA for Procedure Analysis) followed by the message number. The severity code will be either I (Information), W (Warning), E (Error), S (Severe), or U (Terminating). The message text provides a brief explanation of the condition.

In the following example message:

```
IGYGR1421-E Key data-name "REL-KEY" was defined as a windowed date field.
            "REL-KEY" was treated as a non-date in this context.
```

- The message prefix is IGY.
- The compiler phase indicator is GR.

- The message number is 1421.
- The severity code is E.
- The message text is “Key data-name "REL-KEY" was defined as a windowed date field. "REL-KEY" was treated as a non-date in this context.”

IGYGR1411-W *<FILE> <STATUS>* **data-name** *&1* **was defined as a date field.** *&1* **was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the FILE STATUS data name, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler ignores the DATE FORMAT clause on the data item and treats the contents of the field without any special date processing.

IGYGR1413-W *<PASSWORD>* **data-name** *&1* **was defined as a date field.** *&1* **was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the PASSWORD data name, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler ignores the DATE FORMAT clause on the data item and treats the contents of the field without any special date processing.

IGYGR1415-W *ASSIGN USING* **data-name** *&1* **was defined as a date field.** *&1* **was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the ASSIGN USING data name, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler ignores the DATE FORMAT clause on the data item and treats the contents of the field without any special date processing.

IGYGR1416-W *<LABEL> <RECORD>* **data-name** *&1* **was defined as a date field.** *&1* **was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the LABEL RECORD data name, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler ignores the DATE FORMAT clause on the data item and treats the contents of the field without any special date processing.

IGYGR1421-E Key data-name *&1* was defined as a windowed date field. *&1* was treated as a non-date in this context.

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the key data name. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1422-E *<DEPENDING> <ON>* data-name *&1* in the *<RECORD> <VARYING>* clause was defined as a windowed date field. *&1* was treated as a non-date in this context.

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the DEPENDING ON data-item. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1423-W Data-name *&1* was referenced in a *<LINAGE>* clause, but was defined as a date field. *&1* was treated as a non-date in this context.

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item in the LINAGE clause, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler ignores the DATE FORMAT clause on the data item and treats the contents of the field without any special date processing.

IGYDS1424-E The *<DATE> <FORMAT>* clause for item *&1* was not compatible with the *&2* clause. The *<DATE> <FORMAT>* clause was discarded.

Programmer Response: Remove the DATE FORMAT clause from the description of item *&1* . If *&1* is truly a date data item, then consider removing the *&2* clause instead, or remove the DATE FORMAT clause and use the DATEVAL function where *&1* is used in the PROCEDURE DIVISION.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1452-E *<OCCURS> <DEPENDING> <ON>* data-name *&1* was defined as a windowed date field. *&1* was treated as a non-date in this context.

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the OCCURS DEPENDING ON data-item. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1453-E *<ASCENDING>* **or** *<DESCENDING> <KEY> &1* **was defined as a windowed date field.** *&1* **was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use a different data-item as the ASCENDING or DESCENDING KEY data-item. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1454-I A group item contained one or more subordinate date fields.

Programmer Response: None. You might want to make sure that you intended to have date fields in this group item.

System Action: Processing continues.

IGYDS1455-E The length of the *<DATE> <FORMAT>* **pattern did not match the length of group item** *&1***. The** *<DATE> <FORMAT>* **clause was discarded.**

Programmer Response: Change the DATE FORMAT pattern to match the length of the group item, or remove the DATE FORMAT clause.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1456-E Subordinate item *&1* **had a date format that was incompatible with the date format of group item** *&2***. If both items are referenced, results are unpredictable.**

Programmer Response: Change the DATE FORMAT pattern of either *&1* or *&2* so that they are compatible.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1457-W Group item *&1* **was found to consist solely of a date field, but was defined without the** *<DATE> <FORMAT>* **clause.**

Programmer Response: Add a date format clause to *&1*, ignore the message, or remove the DATE FORMAT clause(s) from subordinate items if they are not date data items.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYGR1458-E The condition-name associated with a windowed date field specified a range of values, but the *YEARWINDOW* compiler option specified a sliding window. Execution results are unpredictable.

Programmer Response: The compiler cannot validate that a proper range was specified because the century window will not be known until run time. Either specify a fixed century window with the YEARWINDOW compiler option, or change the VALUE clause for the condition-name.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1459-W Group item *&1* was defined as a date field, but subordinate item *&2* was not defined as a compatible date field.

Programmer Response:

You will receive this message when a subordinate item does not have a DATE FORMAT clause, yet it defines all or part of the year portion of a group date data item. Add a DATE FORMAT clause to the subordinate item. You might have to change the data description so that the subordinate items are compatible with the group item.

If the century field description is separate from the year portion of an expanded date field, ensure that you do not use the century field by itself in a way that is incompatible with using either the expanded version of the group date or the windowed subordinate. To be certain, you can remove the name from the century field. For example:

```
01 CCYYMMDD            DATE FORMAT YYYYXXXX.
   02 CC     PIC 99.
   02 YYMMDD PIC 9(6)  DATE FORMAT YYXXXX.
```

With the example above, message IGYGR1459-W is generated on CC, which might be acceptable to you. You can eliminate the message by removing the `CC` from the data description. You can either replace the name with FILLER, or leave it out.

When the century field is available, use the expanded date version.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYDS1460-I One or more records under *<FD> &1* consisted of or contained a date field.

Programmer Response: None. For information only.

System Action: Processing continues.

IGYDS1461-I *&1* redefined group item *&2*, which contained one or more date fields.

Programmer Response: None. For information only.

System Action: Processing continues.

IGYDS1462-I Group item *&1*, which contained one or more date fields, redefined *&2*.

Programmer Response: None. For information only.

System Action: Processing continues.

IGYDS1463-W Redefining item *&1* and redefined item *&2* were both defined as date fields, but had incompatible *<DATE>* *<FORMAT>* or *<USAGE>* clauses.

Programmer Response: The redefining field (item 1) does not have a DATE FORMAT clause consistent with the redefined field (item 2), or the USAGE clauses cause incompatibilities. Change the DATE FORMAT clauses to be compatible, or remove one of the DATE FORMAT clauses.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYGR1464-E The windowed date value of the first literal was greater than or equal to that of the second literal in the *<THRU>* phrase of the *<VALUE>* clause. The literals were processed as written.

Programmer Response: You can either change the literal values or the YEARWINDOW compiler option specification so that the second literal is greater than the first literal.

If you want the literal values to remain unchanged, change the YEARWINDOW specification. For example, the following statement will compile cleanly with YEARWINDOW(1950):

```
02 END-DATE   PIC 9(6)  DATE FORMAT YYXXXX.
88 END-DATE-RANGE       VALUE 500101 THRU 700101.
```

However, if you specified YEARWINDOW(1960) in the above example, message IGYGR1464 is generated because literal-1 resolves to January 1, 2050 and literal-2 resolves to January 1, 1970.

Alternatively, if your program is dependent on the range value and not the actual date, you can change the literal values so that the first literal resolves to a value less than the second literal based on the YEARWINDOW value. Using the above example, if the compile unit specifies YEARWINDOW(1960), you could change the literal value to:

```
02 END-DATE   PIC 9(6)  DATE FORMAT YYXXXX.
88 END-DATE-RANGE       VALUE 600101 THRU 800101.
```

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1465-W *&1* **redefined** *&2*, **but only one of them had a** *<DATE> <FORMAT>* **clause.**

Programmer Response: Ideally, both &1 and &2 should be date fields, or both should be non-date fields. If the program intends for one to be a date field and one to be a non-date, you can ignore this message. Otherwise, add a DATE FORMAT clause to one or remove the DATE FORMAT clause from the other.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYDS1466-W The area renamed by level-66 item *&1* **contained all or part of a date field.**

Programmer Response: Ideally, both &1 and the renamed area should be date fields, or both should be non-date fields. If the program intends for one to be a date field and one to be a non-date, you can ignore this message. Otherwise, add a DATE FORMAT clause to one or remove the DATE FORMAT clause from the other.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYGR1467-E An alphanumeric literal contained an invalid date value. The date range could not be checked.

Programmer Response: Change the alphanumeric literal into a literal that contains a valid date value.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1468-E A figurative constant was specified in the *<VALUE>* **clause of a condition-name associated with a windowed date field. The condition-name was discarded.**

Programmer Response: Use a literal instead of a figurative constant.

System Action: Compilation continues, but run-time results are unpredictable.

IGYGR1469-E The length of the *<VALUE>* **literal did not match the length of the alphanumeric windowed date conditional variable. The condition-name was discarded.**

Programmer Response: Use a literal that has the same number of characters or digits as the conditional variable.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1476-E A *DATE FORMAT* **clause was found for a windowed date field that was defined with** *<USAGE> BINARY* **but the TRUNC(BIN) option was specified. The** *DATE FORMAT* **clause was discarded.**

Programmer Response: Since windowing assumes that data will be truncated, you cannot have binary windowed dates with TRUNC(BIN), which guarantees no truncation. The solution is to remove the DATE FORMAT clause, and move the date into a PACKED-DECIMAL or DISPLAY date field and use that field in comparisons. Another possibility is to use the TRUNC(OPT) compiler option.

System Action: Compilation continues, but run-time results are unpredictable.

IGYDS1480-W Subordinate item *&1* **was defined as a windowed date field, but group item** *&2* **was defined as an expanded date field.**

Programmer Response: This warning helps identify places in your code that could be using both the expanded version and windowed version of a date. Specifically, this message is generated when the century field is defined separately from the year portion of an expanded date field, as in the following example:

```
01 CCYYMMDD              DATE FORMAT YYYYXXXX.
   02 CC      PIC 99.
   02 YYMMDD  PIC 9(6)   DATE FORMAT YYXXXX.
```

In this case, message IGYGR1480 is generated on YYMMDD, which might be acceptable to you. If so, you need to ensure that the windowed date field (YYMMDD in the example above) is not used by itself in places where windowing might be applied. Otherwise, you could receive inconsistent results since you would be mixing field expansion with windowing for the same date field.

When the century field is available, use the expanded date version (CCYYMMDD in the example above) for the most reliable results.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3245-S Numeric function FUNCTION UNDATE was not allowed in this context. The statement was discarded.

Programmer Response: You will receive this message when a numeric function is used in a MOVE or DISPLAY statement, usually when the UNDATE function is used in a MOVE statement. Only alphanumeric intrinsic functions are allowed in MOVE or DISPLAY statements.

The following example will receive message IGYPA3307-E:

```
01 DATE-FIELD PIC 9(6) DATE FORMAT YYXXXX.
01 NON-DATE   PIC 9(6).
⋮
MOVE DATE-FIELD TO NON-DATE.
```

One solution is to add a DATE FORMAT clause to NON-DATE.

Alternatively, you can replace the MOVE statement with a COMPUTE statement that uses the UNDATE intrinsic function, as shown in the example below:

```
COMPUTE NON-DATE = FUNCTION UNDATE(DATE-FIELD).
```

Note: Simply using the UNDATE function in the MOVE statement will generate message IGYPA3245-S, since numeric functions are not allowed in MOVE statements.

System Action: The statement was discarded from the compilation.

IGYPA3283-S The second argument *&1* for function *DATEVAL* was not an alphanumeric literal, or contained an invalid date pattern. The statement was discarded.

Programmer Response: Specify a valid date pattern in the form of an alphanumeric literal as the second argument for the DATEVAL function. See the description of the DATEVAL function in the Language Reference Manual for more information.

System Action: The statement was discarded from the compilation.

IGYPA3284-S Argument *&1* for function *UNDATE* was not a date field. The statement was discarded.

Programmer Response: Remove the UNDATE function, or if *&1* should have been identified as a date field, add a DATE FORMAT clause to its description.

System Action: The statement was discarded from the compilation.

IGYPA3285-S Argument *&1* for function *DATEVAL* was not alphanumeric or integer. The statement was discarded.

Programmer Response: Remove the DATEVAL function. If *&1* really contains date data, then consider changing its description or moving it to a data item that is alphanumeric or integer and that does have a DATE FORMAT clause. Use that date field in place of *&1* .

System Action: The statement was discarded from the compilation.

IGYPA3286-W Windowed date field &1 was found as an ASCENDING or DESCENDING KEY in the &2 statement, but was not a year-only date field. &1 was treated as a non-date in this context.

Programmer Response: Because windowed SORT/MERGE order is not supported for BINARY dates that have one or more Xs in the date pattern, do one of the following:

- Change the program logic to handle non-windowed order for the result of the SORT or MERGE.
- Change the records to have a separate year field in binary.
- Change the records to have the date field key as another data type, such as external decimal, packed decimal, or alphanumeric.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3287-E A date field result was assigned to year-last date field &1. Execution results are unpredictable.

Programmer Response: Assigning a (non-year-last) date arithmetic result to a year-last date field is not allowed. Check the program logic and correct the date format of the receiving field, or the arithmetic result, to give a meaningful operation.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3288-E Year-last date field &1 was found in an arithmetic operation or expression. Execution results are unpredictable.

Programmer Response: Year-last dates are not allowed in arithmetic expressions since incorrect results could occur. For example, adding 1 to a date defined as 123199 and getting 123200 instead of 123100.

Change the program logic to avoid the arithmetic operation, or use FUNCTION UNDATE on the date field and the COMPUTE statement to perform the arithmetic. When using FUNCTION UNDATE and the COMPUTE statement, the arithmetic is not date sensitive.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3289-E A year-last date field was found in a comparison that did not use EQUAL or NOT EQUAL as the relational operator. Execution results are unpredictable.

Programmer Response: In general, it is incorrect to compare year last dates with relational operators other than EQUAL or NOT EQUAL because the year would have the lowest precedence, and months or days would compare with more importance. For example, 123197 would be greater than 110198 or 123099. Windowing is not done on year-last comparisons for this reason.

Use year-first dates for these types of comparisons, or use separate comparisons for the year and non-year parts of the date.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3290-I The statement contained date logic.

Programmer Response: None. For information only.

System Action: Processing continues.

IGYPA3291-W Windowed date field *&1* was found in a *CICS CALL* statement. It was treated as a non-date in this context.

Programmer Response: Although it is better to pass 4-digit year dates between programs, you can ignore this message if correct date processing will occur. To avoid this message, use a non-date in the CICS statement in place of *&1*.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3292-W Windowed date field *&1* was found in the *<PROCEDURE> <DIVISION>* header. It was treated as a non-date in this context.

Programmer Response: Although it is better to pass 4-digit year dates between programs, you can ignore this message if correct date processing will occur. To avoid this message, use a non-date in the PROCEDURE DIVISION header place of *&1*.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3293-S The length of argument *&1* for function *DATEVAL* did not match the length of the specified date pattern. The statement was discarded.

Programmer Response: Change the date pattern argument for DATEVAL to match *&1.*

System Action: The statement was discarded from the compilation.

IGYPA3294-S Date field *&1* was found as an argument for function *DATEVAL*. The statement was discarded.

Programmer Response: Remove the function DATEVAL, or if *&1* should not be identified as a date, remove the DATE FORMAT clause from its description.

System Action: The statement was discarded from the compilation.

IGYPA3295-W Windowed date field *&1* was found as a receiver in an *<ACCEPT>* statement that did not specify *<DATE>* or *<DAY>*. A non-date value will be transferred to *&1*.

Programmer Response: Ensure that a valid date value is entered as a response to the ACCEPT prompt at execution time. You may need to add code to validate the data that was entered. One way to do this and resolve this warning message is to ACCEPT the data into a non-date field, validate that a date was entered, and then MOVE the data into a date field using the DATEVAL function. You may also ignore this warning if you are confident that correct date data will be entered as a response to the ACCEPT statement.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3296-E Non-date *&1* **was found as a receiver in an** *<ACCEPT> <DATE>* **or** *<DAY>* **statement. Execution results are unpredictable.**

Programmer Response: Add a DATE FORMAT clause to the definition of the non-date.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3297-W Windowed date field *&1* **was found in a** *<DISPLAY>* **statement. It was treated as a non-date in this context.**

Programmer Response: This is a warning that the value that is displayed will not have a century displayed with it, so the year may not be obvious to whoever reads that displayed value. You can ignore this message, or if you want to avoid this warning, move the windowed date field to an expanded date field and DISPLAY that value, use FUNCTION UNDATE on the windowed date field, or use a non-date in the DISPLAY statement instead of the date field.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3298-W The arithmetic result had no date attribute, and was treated as if it had the same date format as receiver *&1***, but with a base year of 1900.**

Programmer Response: If this result should be windowed with the base year specified in the YEARWINDOW option,then use the DATEVAL function on the entire expression. Otherwise, use UNDATE on the date fields in the expression. If windowing the result with the base year 1900 is what you want, you can ignore the message.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3299-I The *<SIZE> <ERROR>* **imperative statements will be executed if the year part of the date result is not valid for windowed date receiving field** *&1***.**

Programmer Response: None

System Action: Processing continues.

IGYPA3300-E Windowed date field *&1* **was found in a** *&2* **statement. It was treated as a non-date in this context.**

Programmer Response:

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3301-E Windowed date field *&1* was found as a reference modification length or position. Execution results are unpredictable.

Programmer Response: Use an expanded date field or non-date in place of *&1*. If *&1* should not have a DATE FORMAT clause, remove it.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3302-E Date field *&1* was found as an operand of an arithmetic operation for which date fields are not supported. Execution results are unpredictable.

Programmer Response: If *&1* is correctly identified as a date field, then use the UNDATE function. Since intrinsic functions are not allowed in arithmetic verbs such as ADD and DIVIDE, you might need to change the verb to a COMPUTE. For example:

Example 1:
```
MULTIPLY YY BY 7 GIVING STRANGE-RESULT.
```

Change to:
```
COMPUTE STRANGE-RESULT = FUNCTION UNDATE(YY) * 7.
```

Example 2:
```
DIVIDE YY BY 4 GIVING QUOTIENT REMAINDER LEAP-YEAR.
```

Change to:
```
COMPUTE LEAP-YEAR = FUNCTION REM (FUNCTION UNDATE(YY),4).
```

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3303-I The windowed date arithmetic result will be expanded and stored into *&1*.

Programmer Response: None.

System Action: Processing continues.

IGYPA3304-W Non-date *&1* was treated as if it had the same date format as date field *&2*, but with a base year of 1900.

Programmer Response: Ideally both &1 and &2 should be date fields or they should both be non-date fields. If &2 is truly a date field, then add a DATE FORMAT clause to the description of &1 or use the DATEVAL function for &1. If &2 is not really a date, then remove the DATE FORMAT clause from its description.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3305-E Date fields *&1* and *&2* had incompatible date formats. Execution results are unpredictable.

Programmer Response: Change the description of &1 or &2 so that they have the same date pattern, or replace one of them with a date field that matches the other.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3306-W Non-date *&1* will be moved to windowed date field *&2*.

Programmer Response: Ideally both &1 and &2 should be date fields or they should both be non-date fields. If &2 is truly a date field, then add a DATE FORMAT clause to the description of &1 or use the DATEVAL function for &1. If &2 is not really a date, then remove the DATE FORMAT clause from its description.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3307-E Date field *&1* was found as a sender, but the receiver was non-date *&2*. *&1* was treated as a non-date in this context.

Programmer Response: Both &1 and &2 should be date fields or they should both be non-date fields. If &1 is not a date, remove the DATE FORMAT clause from its description.

If &1 is truly a date field, then add a DATE FORMAT clause to the description of &2 or use the UNDATE function for &1.

Example: With the following code, you use a MOVE statement (such as `MOVE YY to NONDATE`):

```
77 NONDATE PIC 9(2).
77 YY      PIC 9(2) DATE FORMAT YY.
```

If you want to use FUNCTION UNDATE, use a COMPUTE statement instead of the MOVE statement (such as `COMPUTE NONDATE = FUNCTION UNDATE (YY).`). The MOVE statement is not valid because FUNCTION UNDATE with a numeric argument cannot be used in a MOVE statement.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3308-E Date fields &1 and &2 had different date formats. Execution results are unpredictable.

Programmer Response: Since automatic windowing is not supported for year-last date fields, operations between windowed and expanded dates of the same general type are not supported. For a valid comparison, both year-last date fields must have identical date formats.

For example, the following comparison is invalid: `YYMMDD` and `YYYYMMDD`. Although they have the same number of non-year characters (which is acceptable for year-first date format processing), for year-last processing, the number of year digits must also be identical.

For comparisons of windowed and expanded year-last date fields, you can either manually expand the windowed field or compare the year and non-year parts separately. To do this, declare the year-last date fields as group items whose year components are defined as year-only subcomponents.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3309-I The windowed date value in *&1* will be expanded and moved into *&2*.

Programmer Response: None.

System Action: Processing continues.

IGYPA3310-I A windowed date comparison will occur between *&1* and *&2*.

Programmer Response: None.

System Action: Processing continues.

IGYPA3311-E Windowed date field *&1* was found as a function argument. It was treated as a non-date in this context.

Programmer Response: Either use an expanded date field, use UNDATE, or if *&1* is not really a date data item, remove the DATE FORMAT clause from its description.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3312-E A date arithmetic result was found as a sender, but the receiver was non-date *&1*. Execution results are unpredictable.

Programmer Response:

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3313-E The date format of the arithmetic result was incompatible with the date format of receiving field *&1*. Execution results are unpredictable.

Programmer Response: Change *&1* to match the format of the arithmetic result, or change the arithmetic result to get the date format that matches *&1*.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3314-E Windowed date field *&1* was reference modified. It was treated as a non-date in this context.

Programmer Response: Preferably, use a named subordinate of the windowed date field instead of reference modification.

Optionally, if *&1* is alphanumeric and not a receiver, use the UNDATE function to allow the reference modification. When using UNDATE, no date processing occurs. To enable date processing, use the DATEVAL function with UNDATE, as shown in the example below:

Original Code:
```
IF YYMMDD (1:2) < YY THEN
```

With UNDATE:
```
IF FUNCTION UNDATE (YYMMDD) (1:2) < YY THEN
```

With UNDATE and DATEVAL:
```
IF FUNCTION DATEVAL (FUNCTION UNDATE (YYMMDD) (1:2), 'YY') < YY THEN
```

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3315-W Non-date *&1* did not have the same number of digits as date field *&2*.

Programmer Response: Change the non-date *&1* to have the same number of digits as *&2*, or use the DATEVAL function to make it a date field.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3316-I The *<ACCEPT>* statement will transfer a windowed date value into *&1*.

Programmer Response: None

System Action: Processing continues.

IGYPA3317-E The date format of *&1* was not identical to the date format of the *<ACCEPT>* statement. Execution results are unpredictable.

Programmer Response: Either change the receiver *&1* to match the ACCEPT statement, use a different receiver, or use a different ACCEPT statement.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3318-W Group *&1* contains a windowed date data item, but windowed date processing will not be performed for this non-date group item.

Programmer Response: Use all of the individually named data items in this statement, rather than the group name. That way the compiler can handle any necessary windowed date processing for you.

System Action: Compilation continues, but run-time results may not be year 2000 ready.

IGYPA3319-E Windowed date field *&1* was found as a subscript. It was treated as a non-date in this context.

Programmer Response:

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3320-W Expanded date field *&1* **was reference modified. It was treated as a non-date in this context.**

Programmer Response: Check to see if *&1* should have a DATE FORMAT clause in its description. If so, then use the whole year field with its own name, or simply ignore the message. If *&1* should not be a date field, then remove the DATE FORMAT clause from its description.

System Action: The compiler has made an assumption about a date field or non-date because of inadequate information coded in the program, or to indicate the location of date logic that should be manually checked for correctness. Compilation proceeds, with any assumptions continuing to be applied.

IGYPA3321-E Windowed date field *&1* **was found as an** *<ASCENDING>* **or** *<DESCENDING>* *<KEY>* **in the** *&2* **statement.** *&1* **was treated as a non-date in this context.**

Programmer Response: This message applicable only to workstation programs. You can ignore this message if this program is intended for host execution.

If this program is targeted for execution on the workstation, use INPUT and OUTPUT procedures to expand the date fields on RELEASE and contract them again on RETURN.

Optionally, you can expand the dates in the file and use expanded date fields as the sort/merge keys.

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3322-I Windowed date field *&1* **was found as an** *<ASCENDING>* **or** *<DESCENDING>* *<KEY>* **in the** *&2* **statement. The** *&2* **statement will use a windowed date field sequence for the year portion of this key and any other windowed date field keys.**

Programmer Response: None. This message indicates that either a year field or 00 will sort as greater than 99.

System Action: Processing continues.

IGYPA3323-E *<INDEXED>* **file** *&1* **was found in the** *<GIVING>* **phrase of the** *&2* **statement, but the data item referenced by the first** *<ASCENDING>* *<KEY>* **dataname was a windowed date field. Execution results are unpredictable.**

Programmer Response: Since INDEXED files do not use a windowed date field order when the files are stored, the result of this statement will either be records in a binary collating sequence order (DYNAMIC ACCESS) or in the diagnostic messages from VSAM when COBOL attempts to write key sequenced records out of sequence (SEQUENTIAL ACCESS).

System Action: Compilation continues, but run-time results are unpredictable.

IGYPA3324-E Alphanumeric windowed date field *&1* **was found as an** *<ASCENDING>* **or** *<DESCENDING>* *<KEY>* **in the** *&2* **statement, but the collating sequence used was not** *<EBCDIC>* **or** *<NATIVE>***. The** *&2* **statement will use a windowed date field sequence for the year portion of this key, rather than the collating sequence in effect.**

Programmer Response: You must choose between a windowed date field order using the EBCDIC collating sequence, or a non-windowed date field order (binary order) with a non-EBCDIC collating sequence. Either use a non-windowed date field as the key, or change the collating sequence (either PROGRAM or SORT/MERGE statement collating sequence).

System Action: Compilation continues, but run-time results are unpredictable.

Bibliography

IBM COBOL for OS/390 & VM

Fact Sheet, GC26-9048

Licensed Program Specifications, GC26-9044.

Compiler and Run-Time Migration Guide, GC26-4764

Installation and Customization under OS/390, GC26-9045

Programming Guide, SC26-9049

Language Reference, SC26-9046

Debug Tool User's Guide and Reference, SC09-2137

Diagnosis Guide, GC26-9047

Related Publications for COBOL for OS/390 & VM:

VisualAge COBOL Millennium Language Extensions for OS/390 & VM

Fact Sheet, GC26-9321

Licensed Program Specifications, GC26-9307

Installation and Customization under OS/390, GC26-9045

IBM Language Environment for OS/390 & VM

Concepts Guide, GC28-1945

OS/390 Customization, SC28-1941

Programming Guide, SC28-1939

Programming Reference, SC28-1940

Writing Interlanguage Applications, SC28-1943

Run-Time Migration Guide, SC28-1944

Debugging Guide and Run-Time Messages, SC28-1942

IBM COBOL for MVS & VM

Fact Sheet, GC26-8664

Licensed Program Specifications, GC26-4761

Compiler and Run-Time Migration Guide, GC26-4764

Installation and Customization under MVS, GC26-4766

Programming Guide, SC26-4767

Language Reference, SC26-4769

Debug Tool User's Guide and Reference, SC09-2137

Diagnosis Guide, SC26-3138

Related Publications for COBOL for MVS & VM:

VisualAge COBOL Millennium Language Extensions for MVS & VM

Fact Sheet, GC26-9321

Licensed Program Specifications, GC26-9308

Installation and Customization under MVS, SC26-4766

IBM Language Environment for MVS & VM

Fact Sheet, GC26-4785

Concepts Guide, GC26-4786

Installation and Customization, SC26-4817

Programming Guide, SC26-4818

Programming Reference, SC26-3312

Writing Interlanguage Communications Applications, SC26-8351

Run-Time Migration Guide, SC26-8232

Debugging Guide and Run-Time Messages, SC26-4829

Diagnosis Guide, SC26-3154

Licensed Program Specifications, GC26-4774

Master Index, SC26-3427

IBM COBOL Set for AIX

Fact Sheet, GC26-8484

Language Reference, SC26-9046

Programming Guide, SC26-8423

Program Builder User's Guide, SC09-2201

LPEX User's Guide and Reference, SC09-2202

Getting Started, GC26-8425

IBM VisualAge COBOL

Fact Sheet, GC26-9052

Getting Started on OS/2, GC26-9051

Getting Started on Windows, GC26-8944

Language Reference, SC26-9046

Programming Guide, SC26-9050

Visual Builder User's Guide, SC26-9053

IBM COBOL for VSE/ESA

General Information, GC26-8068

Migration Guide, GC26-8070

Installation and Customization Guide, SC26-8071

Programming Guide, SC26-8072

Language Reference, SC26-8073

Reference Summary, SX26-3834

Diagnosis Guide, SC26-8528

Licensed Program Specifications, GC26-8069

Debug Tool Installation and Customization Guide, SC26-8798

Debug Tool User's Guide and Reference, SC26-8797

Related Publications for COBOL for VSE/ESA:

VisualAge COBOL Millennium Language Extensions for VSE/ESA

Fact Sheet, GC26-9321

Licensed Program Specifications, GC26-9418

Installation and Customization Guide, SC26-8071

IBM Language Environment for VSE/ESA

Fact Sheet, GC33-6679

Concepts Guide, GC33-6680

Installation and Customization Guide, SC33-6682

Programming Guide, SC33-6684

Programming Reference, SC33-6685

Debugging Guide and Run-Time Messages, SC33-6681

Licensed Program Specifications, GC33-6683

Run-Time Migration Guide, SC33-6687

Writing Interlanguage Communication Applications, SC33-6686

C Run-Time Library Reference, SC33-6689

C Run-Time Programming Guide, SC33-6688

Other Related Publications

DFSORT

DFSORT Application Programming Guide, SC33-4035

DFSORT/VSE Application Programming Guide, SC26-7040

COBOL and CICS Command Level Conversion Aid

CCCA for OS/390 & MVS & VM User's Guide, SC26-9400

CCCA for VSE/ESA User's Guide, SC26-9401

Softcopy Publications for OS/390, MVS, VM, and VSE

The following collection kits contain IBM COBOL or related product publications:

OS/390 Collection, SK2T-6700

MVS Collection, SK2T-0710

VM Collection, SK2T-2067

VSE Collection, SK2T-0060

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

We'd Like to Hear from You

COBOL
Millennium Language Extensions Guide

Publication No. GC26-9266-04

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB2VVG at IBMMAIL
  - IBMLink: COBPUBS at STLVM27
  - Internet: COBPUBS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

COBOL
Millennium Language Extensions Guide
Publication No. GC26-9266-04

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

May we contact you to discuss your comments? Yes No

Would you like to receive our response by E-Mail?

_____

Your E-mail address

_____ _____
Name Address

_____ _____
Company or Organization

_____ _____
Phone No.

Readers' Comments
GC26-9266-04

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC26-9266-04

Spine information:

COBOL Millennium Language Extensions Guide