Upgrading to Visual COBOL 2.1 Update 1 for Visual Studio

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

Copyright © Micro Focus 2011-2013. All rights reserved.

MICRO FOCUS, the Micro Focus logo and Visual COBOL are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

2013-04-08

Contents

Upgrading to Visual COBOL for Visual Studio 2010 4
Licensing Changes 4
Resolving Conflicts Between Reserved Keywords and Data Item Names 4
Importing and Scanning Existing COBOL Code 6
Recompile All Source Code 6
Upgrading from Net Express 6
Summary of Differences 7
Backward Compatibility with Earlier Micro Focus Products 10
Backward Compatibility with Previous Versions of Visual Studio 12
Compiling and Building Differences 13
Run-time System Differences 16
Restrictions and Unsupported Features 17
Run-Time Technology Differences 19
Editing and Debugging Differences 20
Tips: Visual Studio IDE Equivalents to IDE Features in Net Express 21
Upgrading from ACUCOBOL-GT 23
Compatibility with ACUCOBOL-GT 23
Upgrading from RM/COBOL® 65
Compatibility with RM/COBOL 65
Native COBOL compared with managed COBOL 124
Customer Feedback 125

Upgrading to Visual COBOL for Visual Studio 2010

This guide provides information on upgrading applications from earlier Micro Focus mainframe development environments to Visual COBOL for Visual Studio 2010. It highlights the differences between the old and new products, and offers solutions on how to keep your application working in the same way as before. The guide also introduces the new concepts and features of the Integrated Development Environment.

Note:

- This documentation uses the name Visual COBOL to refer to Visual COBOL for Visual Studio and Visual COBOL for Eclipse. The full product names are used only when it is necessary to differentiate between the two products.

Benefits of Upgrading

You get a number of important benefits by upgrading to Visual COBOL from earlier Micro Focus development systems or other COBOL systems, such as RM/COBOL and extend® (ACUCOBOL-GT).

Visual COBOL uses a proven industry Integrated Development Environment that supports thousands of clients for developing and deploying critical business applications. Visual COBOL enables unified, collaborative, and cost-effective development through rich, industry-standard tooling and at the same time it helps minimize skills shortages, expands market reach and accelerates time-to-delivery to meet today's agile business requirements.

Licensing Changes

For a number of years Micro Focus used the Micro Focus License Management System for Net Express and Server Express.

Micro Focus now uses a standard industry technology for license management, Sentinel RMS from SafeNet. New product releases use Sentinel RMS, as do updates to existing products.

For more on the Micro Focus Licensing Administration Tool, see *Licensing* in the Visual COBOL help.

Resolving Conflicts Between Reserved Keywords and Data Item Names

Micro Focus continues to enhance the COBOL language, for example, by expanding the list of reserved COBOL words and adding new keywords to it as part of new levels of the COBOL language. Each Micro Focus release corresponds to a particular level. You can use the MFLEVEL Compiler directive to enable Micro Focus-specific reserved words in your code and change the behavior of certain features to be compatible with a specific level of the language.

If you use Visual COBOL to compile applications created with an older Micro Focus product, and these applications use data names that are now reserved keywords in Visual COBOL, you receive a COBOL syntax error COBCH0666 ("Reserved word used as data name or unknown data description qualifier"). To work around this issue and continue using some of the reserved words as data names in your source code, you can either:

- use the REMOVE Compiler directive to remove individual keywords from the reserved words list

- set the MFLEVEL Compiler directive to a lower level which corresponds to the level your applications are at (see the information about MFLEVEL of some Micro Focus products further down this section). This removes all reserved keywords which have been added for levels above that level from the reserved words list.

You can set both directives from the command line, in your source code, or in the **Additional Directives** field in the project's COBOL properties.

Setting directives from the command line

To use REMOVE from aVisual COBOL command prompt, type the following:

```
cobol myprogram.cbl remove(title) ;
```

The command above removes TITLE as a keyword from the language so you can use it as an identifier in a COBOL program.

To use the set of reserved words that was used for Net Express v5.1 WrapPack 5, use this command line:

```
cobol myprogram.cbl mflevel"15" ;
```

Setting directives in the source code

To set either one of the directives in your source code, type the following starting with $ in the indication area of your COBOL program:

```
$set remove "ReservedWord"
```

Or:

```
$set mflevel"nn"
```

Setting directives in the IDE

To set either one of the directives in the project's properties:

1. In the IDE, click **Project** > ***<myproject>*** **Properties** > **COBOL**.
2. Type MFLEVEL"*nn*" or REMOVE "*ReservedWord*" in **Additional Directives**.
3. Click **File > Save All**.

MFLEVEL of some Micro Focus product releases and reserved words added for them

These are the keywords that have been added to the reserved words list for some of the more recent Micro Focus products:

- Visual COBOL R4 (MFLEVEL"16"):

  ATTRIBUTES
  ENCODING
  NAMESPACE
  NAMESPACE-
  VALIDATING
  XML-
  XML-SCHEMA
- Net Express and Server Express versions 6.0 WrapPack 2 and 5.1 WrapPack 5 (MFLEVEL "15"):

  DATA-POINTER
  OBJECT-REFERENCE
- Net Express 6.0 and Server Express 6.0 (MFLEVEL "14"):

  BIT
  BOOLEAN

GROUP-USAGE

Importing and Scanning Existing COBOL Code

You can open a Net Express project from withing Visual COBOL which imports the code and converts it to a Visual Studio COBOL project.

To import projects

The Net Express project format is not the same as the project format in Visual COBOL for Visual Studio 2010 so it is not possible to edit Net Express projects in Visual COBOL directly.

You can open a Net Express project in Visual Studio which invokes the Visual Studio conversion wizard that will convert the project into a Visual Studio one. Click **File** > **Open** > **Project**. The wizard analyzes your Net Express project, converts it to the appropriate project type and sets directives as needed.

To import files

In Visual Studio, you can add COBOL files to a project using the **Add Existing COBOL Items** in Solution Explorer. Visual COBOL imports the files into the project and, if specified in the **Add Existing COBOL Items** wizard, scans the files to determine which files are programs or copybooks and sets the appropriate build actions on them. It also sets the COBOL dialect and EXEC SQL directives as specified in **Tools** > **Options** > **Projects** > **COBOL Project Settings**.

To scan files and set directives

In Visual Studio, to set directives on native COBOL files, use the **Reset COBOL source directives** command in Solution Explorer. This triggers file scanning and sets directives as specified in **Tools** > **Options** > **Projects** > **COBOL Project Settings**.

Recompile All Source Code

Application executables that were compiled using Net Express, RM/COBOL or extend® (ACUCOBOL-GT) must be recompiled from the sources using Visual COBOL.

If you do not recompile, you may receive an error. The exact error depends on the operating system you are running.

You can recompile from the IDE or the command line.

Upgrading from Net Express

You can upgrade COBOL applications that were developed in Net Express to Visual COBOL. The majority of the existing applications will continue to run in Visual COBOL without the need to change their code.

This guide lists the differences between Net Express and Visual COBOL in the following areas:

Compiling and building	Having created a project in Visual COBOL, you can either use the IDE or the command line to build.
Run-time systems	There are some differences between the run-time systems supplied with Visual COBOL and those supplied with Net Express. This, however, will not affect your existing applications and they will continue to run under Visual COBOL - you only need to recompile the applications from the source code with Visual COBOL.

Run-time system technologies	Some technologies behave differently and require some upgrade work.
Restrictions and unsupported features	Some features of Net Express are not available in Visual COBOL. However, there are alternative techniques for many of these features.
Editing and debugging	Much of the Net Express functionality for editing and debugging is available in Visual COBOL, but sometimes with a different name and with a slightly different behavior. In addition there are some new features such as background parsing, which highlights errors as you type and code completion techniques that provide easy access to language elements, enabling you to select and insert them simply.
Visual Studio integration	Visual COBOL is integrated with Microsoft Visual Studio 2010, which provides the functionality to manage projects and debug applications. You can compile your COBOL to native or managed code. Applications previously built in Net Express can be developed and run within the Visual Studio IDE.

Summary of Differences

The majority of the applications created with Net Express will continue to work in Visual COBOL without any changes. However, there are some differences between these development systems you should consider when you upgrade to Visual COBOL.

Compiling and Building Differences

There are several aspects of compiling and building applications that behave differently in Visual COBOL. You might need to change the project properties and update some of the Compiler directives and settings that you previously used.

Output File Formats **on page 13**	Each project compiles into a single file (.dll, .so or .exe), or to multiple files of the same file type with one output file for each source file (.dll, .so, .exe, .int, or .gnt). As well as an .lbr file, which contains a collection of .int and .gnt files on Windows, you now can use a .dll as the container for application components.
Compiler Directives **on page 14**	When you upgrade your source code to Visual COBOL some Compiler directives that were specifically designed for 16-bit systems now produce an error on compilation because they are no longer relevant. You should remove them from your code and directives files before you compile.
Linking **on page 15**	The static run-time system and the single-threaded run-time system on Windows are no longer required and they are not shipped with Visual COBOL. Applications built with Visual COBOL are now linked to the shared or dynamic run-time systems.
Called Programs and Dependencies **on page 15**	At run time, called programs are found in the same way as before. However, there are some new ways to set COBPATH and copy files into a common folder.
File Handler **on page 16**	The File Handler .obj files are not available in Visual COBOL. Visual COBOL uses the File handler packaged in the `mffh.dll` file instead.
SQL Compiler Directive Options **on page 16**	When you upgrade your SQL appplications to Visual COBOL, some applications could require additional SQL compiler directive options to avoid compiler errors.
XML PARSE Statement **on page 16**	In Net Express, the default setting for the XMLPARSE Compiler directive is COMPAT, which causes the XML PARSE statement to return information and

events for IBM Enterprise COBOL Version 3. In Visual COBOL, the default is XMLPARSE(XMLSS), which returns information and events for IBM Enterprise COBOL Version 4.

Run-Time System Differences

There are some differences between the run-time systems supplied with Visual COBOL and those supplied with Net Express and Mainframe Express. These, however, do not affect your existing applications if you recompile them from the source code in Visual COBOL.

OpenESQL **on page 16**	Visual COBOL sets the BEHAVIOR SQL Compiler directive option to MAINFRAME by default to provide optimal performance. To revert to the default behavior exhibited in Net Express, set the BEHAVIOR directive to UNOPTIMIZED.
Single-Threaded Run-Time System **on page 17**	The single-threaded run-time system is not available in Visual COBOL on Windows. Instead, both single-threaded and multi-threaded applications run using the multi-threaded run-time system. This has no effect on your existing applications.
Static-Linked Run-Time System **on page 17**	The static-linked run-time system is not available in Visual COBOL. Instead, you now link native code to the shared or dynamic run-time system. This has no effect on your existing applications.
Visual COBOL Co-existing with Earlier Micro Focus Products **on page 17**	Some additional configuration is required to ensure Visual COBOL and Net Express or Studio Enterprise Edition work properly when installed on the same machine.

Restrictions and Unsupported Features

Some features in earlier Micro Focus products are not available in Visual COBOL. However there are alternative techniques for many of these features.

CBL2XML Utility **on page 17**	The CBL2XML utility is currently available as a command line tool only.
Character-Mode Dialog System **on page 17**	Support for creating character-based user interfaces for applications that run in character environments is available for Visual COBOL if you install the Compatibility AddPack which includes a compatible version of the Character-Mode Dialog System. The AddPack is distributed for free through the *Micro Focus SupportLine Web site*.
COBOL Services as Java and Web Services **on page 17**	COBOL services, such as Java interfaces and Web Services, created with the Interface Mapping Toolkit in Net Express run only under Enterprise Server within Micro Focus Server. They do not run under COBOL Server.
CSBIND **on page 17**	CSBIND is a service package in Net Express, Server Express and Application Server that supports client/server COBOL applications. It is not available in Visual COBOL.
DBMS Preprocessors **on page 18**	Earlier Micro Focus products supported DBMS preprocessor versions that are not supported in Visual COBOL. For a list of currently supported DBMS preprocessors, see the *Database Access Support with Native COBOL* and *Database Access Support with .NET Managed COBOL* topics in your Visual COBOL documentation.
Enterprise Server **on page 18**	Enterprise Server and Server for SOA provide an execution environment for COBOL services and COBOL application programs, including

	mainframe support for CICS, JCL, and IMS. They are not available in Visual COBOL or COBOL Server.
Form Designer on page 18	Form Designer is the Net Express tool for creating user interfaces for CGI-based Internet and intranet applications. Form Designer and the HTML page wizard are not available in Visual COBOL.
FSView on page 18	FSView is a utility for administering Fileshare servers. The FSView GUI is not supported in Visual COBOL.
Host Compatibility Option (HCO) on page 18	Host Compatibility Option (HCO) is not supported in Visual COBOL.
Interface Mapping Toolkit on page 18	The Interface Mapping Toolkit is not supported in Visual COBOL.
INTLEVEL Support on page 18	The INTLEVEL directive is rejected by the Compiler in Visual COBOL.
J2EE Application Servers on page 18	J2EE Application Servers are not supported by Visual COBOL.
NSAPI on page 18	There is no support for NSAPI in Visual COBOL.
Online Help System on page 18	Net Express provided the Online Help System for creating online help from character-based applications, and displaying it on screen. It is not available in Visual COBOL and the Online Help System information file type (.HNF) is not supported.
OO Class and Method Wizards on page 18	The OO Class and Methods wizards are not available in Visual COBOL. However, the run-time components for the base and COM OO class libraries are available.
OpenESQL on page 19	In both Net Express and Studio Enterprise Edition, support is provided for Oracle OCI in OpenESQL. Visual COBOL does not support Oracle OCI in OpenESQL.
Secure Sockets Layer (SSL) on page 19	Secure Sockets Layer (SSL) is a standard mechanism for sending and receiving electronic communications in encrypted form. It is not currently supported in Visual COBOL.
Solo Web Server on page 19	The Solo Web server in Net Express enabled you to debug CGI-based Internet applications on the same machine you used to develop them. It is not available in Visual COBOL.
SQL Option for DB2 on page 19	SQL Option for DB2, also known as XDB, is not supported in Visual COBOL.
Type Library Assistant on page 19	Type Library Assistant is not included in Visual COBOL but the run-time components for the COM and the OO COBOL libraries are still available.
TX Series	The IBM TX Series product used to interface with Websphere in Net Express is not supported in Visual COBOL.
UNIX Publish on page 19	The UNIX Publish feature is superseded by the remote development functionality in Visual COBOL for Eclipse. You use Visual COBOL Development Hub, a remote development server to host your source code and you use the Eclipse IDE on your local machine as the development interface.

Run-Time Technology Differences

Some technologies behave differently in Visual COBOL and this might affect how you upgrade existing applications.

COM Interop **on page 19**	The tools to help create COM objects are not supplied with Visual COBOL. However, the COM run-time components are supplied, so that COM is supported and your applications can interoperate with existing COM objects.
Dialog System **on page 19**	Support for Dialog System applications is available in Visual COBOL for Visual Studio if you install the Compatibility AddPack, distributed for free through the *Micro Focus SupportLine Web site*, and the *Product Updates* section.
File Handling **on page 20**	The way you integrate your own security modules into Fileshare has changed. Also, the FILEMAXSIZE setting is different for Visual COBOL and for Net Express.
Test Coverage **on page 20**	Visual COBOL supports Test Coverage from the command line only.

Editing and Debugging Differences

Much of the edit and debug functionality in Net Express is available in Visual COBOL, but some of it has a different name or slightly different behavior. In addition there are some new features such as background parsing.

Data Tools **on page 20**	The Net Express Data Tools are available as a free AddPack for Visual COBOL for Visual Studio 2010 and 2012.
Debugging Native Object-Oriented COBOL **on page 20**	In Net Express you can examine an object while debugging OO COBOL and display the class that defined the object and also other objects derived from that class. In Visual COBOL, you can also view the class information of native OO COBOL but not while debugging.
Mixed Language Debugging **on page 21**	With Net Express you can debug mixed language applications. Visual COBOL does not support mixed language debugging of native code.
Program Breakpoints **on page 21**	Program breakpoints are breakpoints that stop execution each time a specified program or entry point within the program is called. They are supported in Visual COBOL.
Remote Debugging **on page 21**	The Net Express animserv utility used for debugging programs remotely has been replaced by `cobdebugremote` (or `cobdebugremote64` when debugging 64-bit processes) in Visual COBOL.
Source Pool View **on page 21**	The source pool view in Net Express showed all source files available in the project directory, regardless of whether or not they are used in the current build type. This view is not available in Visual COBOL.

Backward Compatibility with Earlier Micro Focus Products

Backward Compatibility with Net Express and Net Express with .NET 5.1

Default working mode	In versions of Visual COBOL R4 and earlier, the default working mode set by the COBMODE environment variable was 32-bit. With the current release of Visual COBOL and Enterprise Developer, it is 64-bit.
Format of the index files	In Net Express, the default setting of the IDXFORMAT option was 4. With the current release of Visual COBOL, it is 8.

FILEMAXSIZE File Handler configuration option
In Net Express, the default setting for FILEMAXSIZE was 4. With the current release of Visual COBOL, it is 8.

Applications developed using Net Express 5.1 or Net Express with .NET 5.1 might require some changes when you move it to Visual COBOL. In particular, Visual COBOL does not include support for the following functionality that was available in Net Express 5.1:

- Debugging tools:
  - Animator
  - Data Tools (Data File Converter, Data File Editor, Fix File Index, and IMS Database Editor)

    Note: You can separately install the Micro Focus Data File Tools Add Pack which includes the Data File Converter, Data File Editor, and Record Layout Editor. Download the Add Pack from the *Micro Focus SupportLine site*.
  - FSView
  - Remote development
- Runtime/Deployment support:
  - Enterprise Server
  - Single-threaded run-time system
  - Static-linked run-time system
- Programming features:
  - Btrieve
  - ISAPI
  - Mainframe subsystems (CICS, JCL, and IMS)
  - NSAPI

Backward Compatibility with Earlier Versions of Visual COBOL

Calling RM/COBOL compatible library routines
Previously, to call an RM/COBOL compatible library routine, you had to set the DIALECT"RM" Compiler directive, which ensured the correct *call-convention* was used. To set this functionality now, you must explicitly use the correct *call-convention* in the CALL statement.

ILUSING
If you set this Compiler directive using the `$set` command, the imported namespace is only applicable to programs, classes and referenced copybooks in that file. If you set the directive through the IDE or from the command line, the imported namespace is applicable to all programs and classes in the project or specified on the command line.

FLAGCD
This Compiler directive is no longer available in Visual COBOL. Remove it from your code, otherwise you receive a `COBCH0053 Directive invalid or not allowed here` error.

CALLFH
If your code specifies the `ACUFH` parameter, it may now produce adverse effects when used. You should replace it with the methods described in *Configuring Access to Vision Data Files* or *Configuring Access to RM/COBOL Data Files*. Both of these methods offer a fuller-functioning solution to handling these types of data files.

Coexisting with Earlier Micro Focus Products

Run-time system error due to COBCONFIG
A run-time system error occurs if either the COBCONFIG or COBCONFIG_ environment variable is set when you run aVisual COBOL application or when you use Visual COBOL to edit or create projects and the configuration file it refers to contains entries that are not valid for Visual COBOL.

For example, this might happen if you have Net Express or Studio Enterprise Edition installed and either COBCONFIG or COBCONFIG_ is set for it.

To work around this issue, ensure that Visual COBOL is not running and then modify the configuration file by doing one of the following:

- If the invalid tunable is not needed by another application, remove it from the run-time configuration file.
- Add the following as the first line in the configuration file:

  ```
  set cobconfig_error_report=false
  ```
- Unset COBCONFIG (or COBCONFIG_) or set it to another configuration file that does not contain the invalid tunable for the particular session you are running in.

Licensing error due to environment settings

The message "Micro Focus License Manager service is not running" can occur when you invoke a Net Express or Studio Enterprise Edition utility from Visual COBOL. This happens when the tool is invoked with Visual COBOL environment settings while it requires the Net Express or Studio Enterprise Edition ones.

This happens when you edit files such as .dat that have a file association with Net Express or Studio Enterprise Edition. This can also happen when invoking a utility within the Net Express or Studio Enterprise Edition products as an external tool from Visual COBOL.

You can workaround this problem in Visual COBOL as follows:

1. Create a batch file that unsets COBREG_PARSED before the tool is invoked. The batch file contains:

   ```
   Set COBREG_PARSED=
   Call [PathToUtility] %1
   ```

   Where *PathToUtility* is the path to the Net Express or Studio Enterprise Edition utility.
2. In the Visual Studio IDE, add the batch file instead of the utility itself as an external tool.

   This ensures that the proper environment is established when running that tool.

Directives Scan

The default options for directives scan (click **Tools > Options > Projects > COBOL Project Settings** in the IDE) have been changed since the previous release of Visual COBOL. In this release, all options on the **COBOL Project Settings** page are enabled by default.

Backward Compatibility with Previous Versions of Visual Studio

The following sections describe issues that you might encounter when migrating a project created using a Micro Focus product for a version of Visual Studio earlier than Visual Studio 2010.

Case preservation

In the previous version of Visual Studio, you could set the project property **Preserve Case** to false. When you import a project into Visual COBOL for Visual Studio 2010, the directive NOPRESERVECASE is set in the project's properties. This enables teh project to be compiled initially.

When you build with NOPRESERVECASE directive, the following COBOL syntax error can occur:

```
COBCH1094 ("NOPRESERVECASE not supported with ILGEN. Consider removing
NOPRESERVECASE")
```

We recommend that you remove this directive. However, that removing the directive could result in some build errors because the name of some items would be case sensitive and not folded to upper case.

If you choose to keep the directive, be aware that this can cause build issues in the following areas:

- new project items added from a template
- generated code (such as Windows forms, WPF classes, Web forms, and Service references)
- code added from a COBOL snippet
- code added using IntelliSense

Changing the target framework

When changing the target framework in a project, the hint path for the references does not get changed. This means that for some references the previous framework assembly could still be used.

To resolve this, after changing the target framework check the path and runtime version of all the references in the project. You can find this information by right-clicking a reference in Solution Explorer then clicking **Properties**. If any of the references point to an incorrect version, delete and add the reference to the project.

Changes in the template of Web applications

The template for the `MainDetail.aspx` file in the COBOL Web applications has changed between versions 2008 and 2010 of Visual Studio. The `Language` property has been removed, `CodeFile` has been changed to `CodeBehind` and `Inherits` should now include the namespace of the project. As a result, if you try to run or debug managed COBOL Web Applications created with Studio Enterprise Edition in Visual COBOL, you can receive a Compiler error: "829: Could not find method 'Context' with this signature".

To work around this issue, you need to change the `MainDetail.aspx` in your application as follows:

1. In Visual COBOL, open the `MainDetail.aspx` in the editor.
2. Change the first line in the code from:

   ```
   <%@ Page Language="COBOL" AutoEventWireup="true"
   CodeFile="MainDetail.aspx.cbl" Inherits="MainDetail" %>
   ```

   to:

   ```
   <%@ Page AutoEventWireup="true" CodeBehind="MainDetail.aspx.cbl"
   Inherits="namespace.MainDetail" %>
   ```

Compiling and Building Differences

There are several aspects of compiling and building applications that behave differently in Visual COBOL. You might need to change the project properties and update some of the Compiler directives and settings that you previously used.

Output File Formats

Supported file formats - .exe and .dll

Each project compiles into a single file (.dll, .so or .exe), or to multiple files of the same file type with one output file for each source file (.dll, .so, .exe, .int, or .gnt). As well as an .lbr file, which contains a collection of .int and .gnt files on Windows, you now can use a .dll as the container for application components.

Building from the command line

To build a Visual Studio solution from the command line:

1. Click **Start > All Programs > Micro Focus Visual COBOL > Tools > Visual COBOL Command Prompt** to start the Visual COBOL command prompt.
2. From the command prompt, navigate to the project directory.
3. Run the following command to build the solution or the project:

   ```
   MSBuild SolutionName.sln
   ```

   or:

   ```
   MSBuild ProjectName.cblproj
   ```

   To view the MSBuild command line options, execute:

   ```
   MSBuild /?
   ```

Building to multiple output files

Each Visual Studio project compiles into a single file (.dll or .exe).

Instead of an .lbr file, which contained a collection of .int and .gnt files on Windows, you now use a .dll as the container for application components.

Your application can consist of multiple projects, each one building a single output file. To do this, choose from the following techniques:

- Create multiple projects in your solution each one building to either an .exe or a .dll:
  1. Import the source files by adding one file or a collection of source files to a single project.
  2. Configure each project to produce either an .exe or a .dll by setting the **Output type** in **Properties** > ***myProject*** > **Application**.
  3. Build the solution.
- Split your project into multiple projects in your solution each one building to either an .exe or a .dll:
  1. Use the **Create Project from Selection** wizard and split the original project into multiple projects in the same solution.
  2. Move each file to a project of its own.
  3. Configure the projects to produce either an .exe or a .dll, and build the solution.
- Ensure that each project can access any dependent projects, by putting the output files from each project in the same folder.

Compiler Directives

When you upgrade your source code to Visual COBOL some Compiler directives that were specifically designed for 16-bit systems now produce an error on compilation because they are no longer relevant.

The following Compiler directives are no longer relevant and we recommend that you remove them from your code and directives files before you compile:

01SHUFFLE
64KPARA
64KSECT
AUXOPT
CHIP
DATALIT
EANIM
EXPANDDATA
FIXING

FLAG-CHIP
MASM
MODEL
OPTSIZE
OPTSPEED
PARAS
PROTMODE
REGPARM
SEGCROSS
SEGSIZE
SIGNCOMPARE
SMALLDD
TABLESEGCROSS
TRICKLECHECK

Linking

The static run-time system and the single-threaded run-time system on Windows are no longer required and they are not shipped with Visual COBOL. Applications built with Visual COBOL are now linked to the shared or dynamic run-time systems.

Linking from the command line

You can link applications from the Visual COBOL command prompt with the `cbllink` or `cblnames` commands. For example, to produce an `.exe` file, use:

```
cbllink myprogram.cbl
```

To compile and link your code to produce a `.dll` file, use:

```
cbllink -d myprogram.cbl
```

With these commands, the single-threaded and static-linking options are automatically mapped onto the multi-threaded and shared run-time systems respectively.

Linking from the IDE

To specify what to link:

1. Click **Project** > ***myProject*** **Properties**.
2. Click the **COBOL Link** tab on the left-hand side of the **Properties** window and specify your link settings.

Called Programs and Dependencies

At run time, called programs are found in the same way as before. However, there are some new ways to set COBPATH and copy files into a common folder.

To build the called programs

You can build your called programs into your application executable, in which case the called programs are found without any further configuration.

When you build the called programs into a .dll file, you can set a property to store the built .dll files in the same folder as the application executable, provided the application project is in the same solution. To do this:

1. In the same solution as your main application project, create a project for the called programs.
2. In the project's properties, on the Application page, set the **Output type** to **Link Library** (which represents a Dynamic Link Library (.dll)).

3. On the COBOL page, set the **Output path** to the same location as that for the built application .exe file.
4. If you want to debug the .dll file together with the application, on the Debug page, set the **Working directory** to point to the folder containing the built .dll file.
5. Build the project.

To set the COBPATH environment variable

Add the COBPATH environment variable to the application configuration file as follows:

1. Right-click your main project and click **Add** > **New Item** > **Application Configuration File**.
2. Double-click **Application.config** in Solution Explorer.
3. In the **Name** field, specify COBPATH.
4. In the **Value** field, specify the full path of the folder. For example:
   ```
   \users\myPath\
   ```
5. Click **Set**.

File Handler

The File Handler .obj files are not available in Visual COBOL. Visual COBOL uses the File handler packaged in the `mffh.dll` file instead.

If the application you are upgrading from Net Express used the File Handler .obj files, when you link your application in Visual COBOL the linker will emit a warning. The application will continue to operate as before provided that you supply the `mffh.dll` file with it.

SQL Compiler Directive Options

If you get errors in Visual COBOL when compiling an object application that was created in Net Express or Studio Enterprise Edition, recompile specifying the GEN-CLASS-VAR SQL Compiler directive option in addition to other appropriate options.

XML PARSE Statement

In Net Express, the default setting for the XMLPARSE Compiler directive is COMPAT, which causes the XML PARSE statement to return information and events for IBM Enterprise COBOL Version 3. In Visual COBOL, the default is XMLPARSE(XMLSS), which returns information and events for IBM Enterprise COBOL Version 4.

To emulate the Net Express behavior in Visual COBOL, specify the XMLPARSE(COMPAT) Compiler directive option.

For a summary of the differences in event information between XMLPARSE(XMLSS) and XMLPARSE(COMPAT), see the *Special Registers* topic in your Visual COBOL documentation.

Run-time System Differences

There are some differences between the run-time systems supplied with Visual COBOL and those supplied with Net Express and Mainframe Express. These, however, do not affect your existing applications if you recompile them from the source code in Visual COBOL.

The changes in the run-time system are described in the following sections.

OpenESQL

Visual COBOL sets the BEHAVIOR SQL Compiler directive option to MAINFRAME by default to provide optimal performance. To revert to the default behavior exhibited in Net Express, set the BEHAVIOR directive to UNOPTIMIZED.

Single-Threaded Run-Time System

The single-threaded run-time system is not available in Visual COBOL on Windows. Instead, both single-threaded and multi-threaded applications run using the multi-threaded run-time system. This has no effect on your existing applications.

Static-Linked Run-Time System

The static-linked run-time system is not available in Visual COBOL. Instead, you now link native code to the shared or dynamic run-time system. This has no effect on your existing applications.

See *Linking Native COBOL Code* in the product Help.

Visual COBOL Co-existing with Earlier Micro Focus Products

If you have Visual COBOL and Net Express or Studio Enterprise Edition installed on the same machine, you sometimes receive a run-time system error if either the COBCONFIG or COBCONFIG_ environment variable is set when you run aVisual COBOL application the configuration file it refers to contains entries that are not valid for Visual COBOL.

To work around this issue, ensure that Visual COBOL is not running and then modify the configuration file by doing one of the following:

- If the invalid tunable is not needed by another application, remove it from the run-time configuration file.
- Add the following as the first line in the configuration file:

  ```
  set cobconfig_error_report=false
  ```
- Unset COBCONFIG (or COBCONFIG_) or set it to another configuration file that does not contain the invalid tunable for the particular session you are running in.

Restrictions and Unsupported Features

Some features in earlier Micro Focus products are not available in Visual COBOL. However there are alternative techniques for many of these features.

CBL2XML Utility

The CBL2XML utility is currently available as a command line tool only.

Character-Mode Dialog System

Support for creating character-based user interfaces for applications that run in character environments is available for Visual COBOL if you install the Compatibility AddPack which includes a compatible version of the Character-Mode Dialog System. The AddPack is distributed for free through the *Micro Focus SupportLine Web site*.

COBOL Services as Java and Web Services

COBOL services, such as Java interfaces and Web Services, created with the Interface Mapping Toolkit in Net Express run only under Enterprise Server within Micro Focus Server. They do not run under COBOL Server.

To run Web Services with Visual COBOL, you need to recreate the Web Services as managed code.

For more information, check the samples for Web Services available in Samples Browser, and the tutorials in the product Help.

CSBIND

CSBIND is a service package in Net Express, Server Express and Application Server that supports client/server COBOL applications. It is not available in Visual COBOL.

DBMS Preprocessors

Earlier Micro Focus products supported DBMS preprocessor versions that are not supported in Visual COBOL. For a list of currently supported DBMS preprocessors, see the *Database Access Support with Native COBOL* and *Database Access Support with .NET Managed COBOL* topics in your Visual COBOL documentation.

Enterprise Server

Enterprise Server and Server for SOA provide an execution environment for COBOL services and COBOL application programs, including mainframe support for CICS, JCL, and IMS. They are not available in Visual COBOL or COBOL Server.

Form Designer

Form Designer is the Net Express tool for creating user interfaces for CGI-based Internet and intranet applications. Form Designer and the HTML page wizard are not available in Visual COBOL.

FSView

FSView is a utility for administering Fileshare servers. The FSView GUI is not supported in Visual COBOL.

Visual COBOL provides all the FSView functions through the command-line utility `fsview`. For more information see *File Handling Reference* > *FSView* > *FSVIEW Command Line* in the product Help.

Host Compatibility Option (HCO)

Host Compatibility Option (HCO) is not supported in Visual COBOL.

Interface Mapping Toolkit

The Interface Mapping Toolkit is not supported in Visual COBOL.

INTLEVEL Support

The INTLEVEL directive is rejected by the Compiler in Visual COBOL.

An INTLEVEL of 1, 2, or 3 is no longer supported and causes compilation errors. Other values are reserved for internal use and should not be used.

J2EE Application Servers

J2EE Application Servers are not supported by Visual COBOL.

NSAPI

There is no support for NSAPI in Visual COBOL.

Online Help System

Net Express provided the Online Help System for creating online help from character-based applications, and displaying it on screen. It is not available in Visual COBOL and the Online Help System information file type (.HNF) is not supported.

OO Class and Method Wizards

The OO Class and Methods wizards are not available in Visual COBOL. However, the run-time components for the base and COM OO class libraries are available.

In addition, the GUI and OLE class libraries are available in the Dialog System AddPack.

Note: The Compatibility AddPack for Visual COBOL is not part of Visual COBOL or the COBOL Server. It is separately installable and available from the *Product Updates* section on the *Micro Focus SupportLine Web site*.

OpenESQL

In both Net Express and Studio Enterprise Edition, support is provided for Oracle OCI in OpenESQL. Visual COBOL does not support Oracle OCI in OpenESQL.

Secure Sockets Layer (SSL)

Secure Sockets Layer (SSL) is a standard mechanism for sending and receiving electronic communications in encrypted form. It is not currently supported in Visual COBOL.

Solo Web Server

The Solo Web server in Net Express enabled you to debug CGI-based Internet applications on the same machine you used to develop them. It is not available in Visual COBOL.

In Visual COBOL, you need to use Apache2 or IIS servers for the CGI programs you create.

SQL Option for DB2

SQL Option for DB2, also known as XDB, is not supported in Visual COBOL.

Type Library Assistant

Type Library Assistant is not included in Visual COBOL but the run-time components for the COM and the OO COBOL libraries are still available.

TX Series

The IBM TX Series product used to interface with Websphere in Net Express is not supported in Visual COBOL.

UNIX Publish

The UNIX Publish feature is superseded by the remote development functionality in Visual COBOL for Eclipse. You use Visual COBOL Development Hub, a remote development server to host your source code and you use the Eclipse IDE on your local machine as the development interface.

Run-Time Technology Differences

Some technologies behave differently in Visual COBOL and this might affect how you upgrade existing applications.

COM Interop

The tools to help create COM objects are not supplied with Visual COBOL. However, the COM run-time components are supplied, so that COM is supported and your applications can interoperate with existing COM objects.

Documentation about COM Interoperability is available on the *Micro Focus SupportLine Web site* as part of the Net Express 5.1 documentation. See *Programming* > *COM and COBOL* in your product documentation.

Dialog System

Support for Dialog System applications is available in Visual COBOL for Visual Studio if you install the Compatibility AddPack, distributed for free through the *Micro Focus SupportLine Web site*, and the *Product Updates* section.

The Compatibility AddPack for Visual COBOL includes the Dialog System GUI component that enables you to run and modernize Dialog System applications with Visual COBOL. The AddPack enables you to upgrade an application to Visual COBOL and from there, you can run the application without change, or modernize it over time.

The application runs under COBOL Server and the Dialog System run-time system in the Add Pack.

Note: The Compatibility AddPack for Visual COBOL is not part of Visual COBOL or the COBOL Server. It is separately installable and available from the *Product Updates* section on the *Micro Focus SupportLine Web site*.

File Handling

The way you integrate your own security modules into Fileshare has changed. Also, the FILEMAXSIZE setting is different for Visual COBOL and for Net Express.

Using security modules

The way you integrate your own security modules (`fhrdrpwd`, `fsseclog` and `fssecopn`) into Fileshare has changed.

In Visual COBOL, you no longer relink Fileshare but you need to supply your own separate files, which are .dll files. For more information, see *Writing Your Own FHRdrPwd Module*, *File Access Validation Module* and *Logon Validation Module* in the *File Handling* section of your product Help.

To use `fsseclog` and `fssecopn`, you need to link one or both of them into a `cobfssecurity.dll` or a shared object and place on the search path. Fileshare will issue a message indicating that it has loaded user security modules.

Sharing data files between applications built in Visual COBOL and others built using Net Express

If you have applications that access the same data files, all those applications should be built with the same FILEMAXSIZE setting. However, applications built with Visual COBOL use a default setting of FILEMAXSIZE=8 while those built in Net Express use FILEMAXSIZE=4.

In Visual COBOL you need to set the FILEMAXSIZE setting in the file handler configuration file (EXTFH.CFG). This ensures Net Express and Visual COBOL are all using the same setting and that programs running under the Net Express run-time system do not access the same files as programs running under the Visual COBOL run-time system.

Btrieve

Btrieve is the file handling system from Pervasive Software Inc. It is not supported in Visual COBOL.

Test Coverage

Visual COBOL supports Test Coverage from the command line only.

Editing and Debugging Differences

Much of the edit and debug functionality in Net Express is available in Visual COBOL, but some of it has a different name or slightly different behavior. In addition there are some new features such as background parsing.

Data Tools

The Net Express Data Tools are available as a free AddPack for Visual COBOL for Visual Studio 2010 and 2012.

The Micro Focus Data File Tools AddPack includes the Data File Converter, Data File Editor, and the Record Layout Editor.

You can download the Micro Focus Data File Tools AddPack from the *Product Updates* section on the *Micro Focus SupportLine site*.

Debugging Native Object-Oriented COBOL

In Net Express you can examine an object while debugging OO COBOL and display the class that defined the object and also other objects derived from that class. In Visual COBOL, you can also view the class information of native OO COBOL but not while debugging.

Mixed Language Debugging

With Net Express you can debug mixed language applications. Visual COBOL does not support mixed language debugging of native code.

To debug applications that contain programs in different languages, you need to debug the native COBOL and the non-COBOL code separately.

Note that you can debug managed COBOL and other managed languages together seamlessly.

Program Breakpoints

Program breakpoints are breakpoints that stop execution each time a specified program or entry point within the program is called. They are supported in Visual COBOL.

Remote Debugging

The Net Express animserv utility used for debugging programs remotely has been replaced by `cobdebugremote` (or `cobdebugremote64` when debugging 64-bit processes) in Visual COBOL.

To debug locally-developed programs on a remote machine you must start `cobdebugremote` (or `cobdebugremote64` when debugging 64-bit processes) before communication can be established. See the Visual COBOL help for more information on cobdebugremote.

Restriction: You can only remotely debug applications that are running on Windows.

For more information, see the section on *Remote Debugging* in your product help.

Source Pool View

The source pool view in Net Express showed all source files available in the project directory, regardless of whether or not they are used in the current build type. This view is not available in Visual COBOL.

However, similar functionality is available in Visual COBOL, by using the Project Details window, where you can view all files in a project or solution, sort the files by various file details, access the file properties and reset directives on them.

Tips: Visual Studio IDE Equivalents to IDE Features in Net Express

The following table shows Net Express IDE features and their corresponding equivalents and locations in Visual Studio.

Functionality	In Net Express	In Visual COBOL for Visual Studio
Project Control		
Project filename	`*.APP`	`*.cblproj`
Add file to project		Right-click the project in Solution Explorer. Choose **Add > New Item** to create a new file from the supported types in the project directory. To add an existing file, choose **Add > Existing Item** and browse to the location of the file to select it. This adds a link in the project to the file but does not copy it in the project directory. To add existing COBOL files, choose **Add Existing COBOL Items**.
Copybook path		Choose **Project >** ***project*****Properties** and select the **Copybook paths** tab.

Functionality	In Net Express	In Visual COBOL for Visual Studio
Build settings for the project: • COBOL • Preprocessor • Additional Directive		Click **Project > *project* Properties**, go to the **COBOL** tab and choose a configuration in the **Configuration** field. To create a new build configuration or to edit one, click **Build > Configuration Manager**.
Execution environment settings: • General • COBOL		The execution environment is COBOL Server.
Debug settings: • DateWarp • Stored Procedures		
Editing		
Suggest Word/Content Assist	**CTRL+G**	**CTRL+Space**
Locate	**F12** (or context menu **Locate**)	**F12**
COBOL Find	**CTRL+Shift+F12** (or context menu **COBOL Find**)	**Shift+F12**
Compress	Tool bar compress (or context menu **Compress**)	
Bookmark	**CTRL+F2**	**CTRL+B, T**
Compiling		
Single file Compile	**CTRL+F7** (or click **check mark**)	In **Solution Explorer**, right-click the file you want to compile and click **Compile**. **Note:** This applies to native code only.
Build	**F7** (or click **build**)	
Build All	**ALT+B A**	Click **Build > Build *<project>***.
Debugging		
Start Debugging	**Alt+D A**	Choose **Debug > Start Debugging** or press **F5**.
Stop Debugging	**Shift+F5**	Choose **Debug > Stop Debugging**.
Restart Debugging	**Ctrl+Shift+F5**	
Run	**F5**	**F5**
Step	**F11** (or click **step**)	**F11**
Step All	**Ctrl+F5**	
Run Thru		
Run Return		
Run to Cursor	**Shift+F10** (or context menu)	**Ctrl+F10**
Skip to Cursor	**CTRL+Shift+F10**	context menu
Skip Statement		

Functionality	In Net Express	In Visual COBOL for Visual Studio
Skip Return		
Examine ' *data item'*	**Shift+F9**	**Shift+F9**
Breakpoint set	**F9**	Double-click in the left margin of editor next to the a line of code, or right-click the line and choose **Breakpoint > Insert Breakpoint**, or press **Shift+F9**.
Conditional Breakpoint		**Breakpoint > Condition**
Break on Data Change	Via list view	You can break on data change in native COBOL projects, by right-clicking and choosing **Add COBOL Watchpoint**.
Attach to Process		Click **Debug > Attach to Process**, or **Ctrl +Alt+P**
Just-In-Time Debugging		Click **Tools > Options > Debugging > Just-In-Time Debugging**, and check **Micro Focus Native Debugger**. **Note:** This applies to native code only.

Upgrading from ACUCOBOL-GT

There are conceptual and behavioral differences between Visual COBOL and ACUCOBOL-GT, part of the Micro Focus extend® product family, and these differences can affect the way you upgrade existing applications to Visual COBOL.

Refer to the *Compatibility with ACUCOBOL-GT* section for guidance and best practice on moving your applications to Visual COBOL. It covers:

- Supported ACUCOBOL-GT features, including detailed information on support for compiler options and standard library routines.
- Syntactical differences between the two COBOL dialects, including workarounds or equivalent syntax where applicable.
- Detailed support of compatible ACUCOBOL Windowing syntax.
- Details on how to configure your applications to continue using your Vision data files.

Compatibility with ACUCOBOL-GT

The following sections describe supported ACUCOBOL-GT features and how to enable them.

Converting ACUCOBOL-GT Applications

With Visual COBOL you can build, compile and debug ACUCOBOL-GT applications. Certain Compiler directives are provided to enable compatibility with some of ACUCOBOL-GT's language extensions, data files, and other behaviors.

There is also a modernization tool available that helps to locate and transform incompatibilities in your ACUCOBOL-GT sources, making it compliant in this COBOL system. This tool is available as an AddPack; see the *Product Updates* section of the SupportLine website (*http://supportline.microfocus.com*) for details on the *ACUCOBOL-GT to Visual COBOL Modernization* AddPack.

After you have converted your application, you must run, license, and distribute your programs in the same manner as other Micro Focus programs. There is currently no clone of the ACUCOBOL-GT runtime known as `wrun32`.

This section describes the ACUCOBOL-GT compatibility features, such as how they are enabled within Visual COBOL, and also the potential problems you may encounter with some aspects of the converted source code.

Enabling ACUCOBOL-GT Compatibility

Compile your ACUCOBOL-GT source code with certain Compiler directives that enable support for ACUCOBOL-GT syntax, data types, and other behaviors. A number of traditional ACUCOBOL-GT compiler options are also available.

Compiler Directives for ACUCOBOL-GT Compatibility

There are a number of Compiler directives that provide compatibility with ACUCOBOL-GT. Use the `DIALECT"ACU"` directive to set all of these directives at once.

By setting `DIALECT"ACU"` you enable certain reserved words, data type storage behavior, and more. See *ACU DIALECT setting* for full details of the directives that are set.

You can set this directive in your source code directly, through the COBOL project options interface in your IDE or from the command line.

Compiler Option Support

You can use many of the ACUCOBOL-GT compiler options when compiling, by setting then with the ACUOPT Compiler directive. A list of the supported options is listed in this section.

Alternatively, you can compile using a clone of the ACUCOBOL-GT compiler known as `ccbl.exe` . This executable is located in the `bin` directory found here: `%ProgramFiles(x86)%\Micro Focus\Visual COBOL`. `ccbl.exe` compiles to `.int` code unless you specify one of the ccbl's native code options, in which case it produces `.gnt` code.

Setting Compiler Options

You can set the ACUCOBOL-GT compiler options by using the ACUOPT Compiler directive, or from a command line utility.

This COBOL system supports many of the compiler options available with the ACUCOBOL-GT (Acu) compiler. To specify these options use the ACUOPT Compiler directive along with the traditional ACU compiler option name.

For example:

```
ACUOPT(-option)
```

Or:

```
ACUOPT(--option)
```

ACUOPT automatically sets the ACU directive, which turns on overall ACUCOBOL-GT compatibility.

NOACU or NOACUOPT are not allowed.

Alternatively, you can compile using a clone of the ACUCOBOL-GT compiler known as `ccbl.exe` . This executable is located in the bin directory found here: `%ProgramFiles(x86)%\Micro Focus\Visual COBOL`. `ccbl.exe` compiles to `.int` code unless you specify one of the ccbl's native code options, in which case it produces `.gnt` code.

Supported ACUCOBOL-GT Compiler Options

There are a number of ACUCOBOL-GT Compiler options supported in Visual COBOL, which you enable using the ACUOPT Compiler directive.

General Support Notes

Visual COBOL supports the following ACUCOBOL-GT compiler functionality:

- Grouping of options
- CBLFLAGS environment variable
- Replacement of @ by the base name of the source file.

The following compiler options are available:

Standard Options

The standard options enable you to control certain compile time options, such as verbose output and renaming the object file.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-e	This option must be followed by a file name (as the next separate argument). When specified, this option causes the error listing to be written to the specified file instead of the screen. This file is removed if no errors are found.
-o	This option must be followed by a file name (as the next separate argument), which becomes the name of the object file instead of `source-name.int`. This file is removed if the compiler detects errors in the source.
-v	This option has multiple applications: • If it is the first and only option on the command line, then the compiler runs in "Version" mode. Using `-v`, you can display version information, the copyright notice, and other information. • Otherwise, if it is used in combination with other options, it causes the compiler to be verbose about its progress. Because `-v` is the lead-in sequence for the video options, this option should be specified by itself.
-w	This option causes warning messages to be suppressed (a warning condition is never a fatal compilation error). Suppressing warning messages can be helpful when you are converting programs from another COBOL dialect that uses slightly different syntaxes.
-x	This causes the CBLFLAGS environment variable to be ignored.

Native Object Code Options

The native object code option enables you to execute object files that contain native instructions for select families of processors.

The following compiler option is supported in Visual COBOL when using `ccbl` from the command line.

Option	Definition
-n	The Compiler produces native code specific to the bit arrangement and local machine. There is no support for cross-generation of native code.

Listing Options

The listing options enable you to control listing information generated with an object file.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Note: The results of these options will differ from how they appear in ACUCOBOL-GT COBOL, as they map to listing Compiler directives in Visual COBOL.

Option	Definition
-La	This option maps to the ERRLIST Compiler directive.
-Lc	This option maps to the XREF and RESEQ Compiler directives.
-Lf	This option maps to the COPYLIST Compiler directive.
-Li	This option maps to the ERRLIST Compiler directive.
-Ll	This option maps to the FORM Compiler directive.
-Lo	This option maps to the LIST Compiler directive.
-Ls	This option maps to the DATAMAP Compiler directive.
-Lw	This option maps to the LISTWIDTH Compiler directive.

Internal Table Options

The Internal Table options available in ACUCOBOL-GT are not required in Visual COBOL. The following options are accepted by the compiler, but are ignored.

Option	Definition
-Td	Identifier and statement table — sets the maximum number of items in each statement. The default value is 4096.
-Te	Subscript statement table — sets the maximum size for OCCURS statements. The default value is 256.

Compatibility Options

The compatibility options enable you to control the compatibility with certain other COBOL systems.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Ca	This option causes simple ACCEPT and DISPLAY statements to be treated in accordance with ANSI semantics. Specifying this option is the same as specifying FROM CONSOLE for all simple ACCEPT statements and UPON CONSOLE for all simple DISPLAY statements. You can control this behavior for individual ACCEPT or DISPLAY statements by specifying an explicit FROM/UPON phrase.
-Ci	This option sets the compiler to be compatible with ICOBOL for certain COBOL constructs.

Option	Definition
-Cr	This option sets the compiler to RM/COBOL compatibility mode.
-Cv	This option sets the compiler to IBM DOS/VS compatibility mode.

Source Options

The source options enable you to modify the way that the Compiler treats the physical source files.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Sa	This causes the compiler to assume that the input source is in the standard ANSI source format.
-Sd	Setting this option causes debugging lines marked with `D` in the indicator area to be treated as normal source lines instead of comment lines. This is equivalent to supplying the phrase `WITH DEBUGGING MODE` in the SOURCE-COMPUTER paragraph.
-Sp	With this option you can specify a series of directories to be searched when the compiler is looking for COPY libraries. This option is followed (as the next separate argument) by the set of directories to search.
-St	This option forces the compiler to use the terminal source format.
-S1..-S9	Specifying a digit with `-S` uses alternate tab stops in source files. When this option is used, tabs will be set every *#* columns apart, where *#* is the number specified. For example, `-S4` will set tab stops at every fourth column. Tab stops always start in column 1.

Reserved Word Options

The reserved word options enable you to override the behavior of reserved words and synonyms.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Rc	Allows you to change a reserved word. This option must be followed by two separate arguments: The first is the reserved word you want to change. The second is the word that you want to use instead. For example, `-Rc TITLE NAME` will allow you to use "TITLE" as a user-defined word and will cause the word `NAME` to be treated as the reserved word `TITLE`. You may not specify a word that is already reserved as the new reserved word. This option may be repeated to transform multiple reserved words.

Option	Definition
-Rn	Allows you to make a reserved word a synonym for another reserved word. This option must be followed by two separate arguments: The first is the reserved word for which you want a synonym. The second is the word that functions as the synonym. For example, `-Rn COMP COMP-5` causes `COMP-5` to be treated the same as the reserved word `COMP`. This option may be repeated to make multiple synonyms.
-Rw	This option allows you to suppress a particular reserved word. The option must be followed (as the next separate argument) by the reserved word you want to suppress. This option may be repeated to suppress multiple reserved words. This option also allows you to suppress some non-reserved words, such as control names (for example, `entry-field` and `label`) or property names (for example, `max-text` and `bitmap-number`).

Data Storage Options

The data storage options control the behavior of certain data items and how they are stored.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-D1	This option causes any data item whose underlying type is binary to be stored in one byte if that data item has only one or two digits. Normally, such a data item would be stored in two bytes.
-D2	This option causes COMPUTATIONAL data items to be treated as if they were declared as COMPUTATIONAL-2. This is the default when you are using RM/COBOL compatibility mode.
-D5	This option causes data items declared as BINARY to be treated as if they were declared as COMPUTATIONAL-5. This causes the values to be stored in the host machine's native byte-ordering instead of the machine-independent byte-ordering normally used. This option should be used with caution, because it can lead to programs that are not portable.
-D6	This option causes unsigned data items declared as PACKED-DECIMAL to be treated as if they were declared as COMPUTATIONAL-6. This saves one-half of a byte because the compiler will not generate any storage for the sign.
-D7	This option allows you to match one of the binary storage conventions used by Micro Focus COBOL. That convention is identical to the ACUCOBOL-GT `-Dm`

Option	Definition
	convention, except that a PIC 9(7) data item (unsigned) is stored in 3 bytes instead of 4 and a PIC 9(12) data item (unsigned) is stored in 5 bytes instead of 6. When you use this option, the size of a binary item is determined as follows (the value in the table is the number of bytes occupied by the data item): **Number of Signed** **Unsigned** **9's in PIC Storage** **Storage** 1 - 2 1 1 3 - 4 2 2 5 - 6 3 3 7 4 3 8 - 9 4 4 10 - 11 5 5 12 6 5 13 - 14 6 6 15 - 16 7 7 17 - 18 8 8
-Da	This allows you to specify the data alignment modulus for level 01 and level 77 data items. Normally, level 01 and level 77 data items are aligned on a 4-byte boundary (modulus 4). This is optimal for 32-bit architectures. You can specify an alternate alignment boundary by following this option with the desired modulus. This should be specified as a single digit that immediately follows the `-Da` as part of the same argument. For example, `-Da8` specifies that data should be aligned on 8-byte boundaries, which can provide improved performance on a 64-bit machine.
-Db	This causes COMPUTATIONAL data items to be treated as if they were declared as BINARY data items. This is the default when you are using VAX COBOL compatibility mode.
-DCa	This selects the ACUCOBOL-GT storage convention. It is the default setting. This convention is also compatible with data produced by RM/COBOL (not RM/COBOL-85) and previous versions of ACUCOBOL-GT. It also produces slightly faster code.
-DCb	This selects the MBP COBOL sign storage convention.Note that the MBP COBOL sign storage convention for USAGE DISPLAY directly conflicts with that used by IBM COBOL and some other COBOLs. As a

Option	Definition
	result, signed USAGE DISPLAY items in the MBP format are correctly understood only when the program is compiled with `-Dcb`. This is unlike the other sign conventions in which the runtime can usually extract the correct value even when a mismatched sign convention is specified at compile time. Also note that MBP COBOL does not have the COMP-2 storage type. The convention that ACUCOBOL-GT implements (Positive: X"0C"; Negative: X"0D") was chosen because MBP COBOL most closely matches the sign storage of other COBOLs that use that convention.
-DCi	This selects the IBM storage convention. It is compatible with IBM COBOL, as well as with several others including RM/COBOL-85. It is also compatible with the X/Open COBOL standard.
-DCm	This selects the Micro Focus storage convention. It is compatible with Micro Focus COBOL when the Micro Focus ASCII sign-storage option is used (this is the Micro Focus default).
-DCn	This causes a different numeric format to be used. The format is the same as the one used when the `-Dci` option is used, except that positive COMP-3 items use `X"0B"` as the positive sign value instead of `X"0C"`. This option is compatible with NCR COBOL.
-DCr	This selects the Realia sign storage convention. Sign information for S9(*n*) variables is stored using the conventions for Realia COBOL, and their conversion to binary decimal is the same as that performed by the Realia compiler.
-DCv	This creates numeric sign formats that are compatible with VAX COBOL. These are identical to the IBM formats, except that unsigned COMP-3 fields place `X"0C"` in the sign position, instead of `X"0F"`. The ANSI definition of COBOL does not state how signs should be stored in numeric fields (except for the case of SIGN IS SEPARATE). As a result, different COBOL vendors use different conventions. By using the options `-Dca`, `-Dci`, `-Dcm`, `-Dcn`, or `-Dcv`, you may select alternate sign-storage conventions. Doing so is useful in the following cases: • If you need to export data to another COBOL system and need to match its sign-storage convention. • If you are importing data from another COBOL system, and that data contains key fields with signed data. Keys are treated alphanumerically, so if you use the incorrect sign-storage convention, ACUCOBOL-GT will not find a matching key when it is doing a READ.

Option	Definition
	The storage-convention affects how data appears in USAGE DISPLAY, COMP-2, and COMP-3 data types.
-Dd31	This option supports data items with up to 31-digits or 16 bytes. When this option is in effect, you may use as many as 31 `X` or `9` symbols in a PIC, instead of the usual 18. The maximum number of bytes in a COMP-X or COMP-N data item, whose picture contains only "X" symbols, is 16, instead of the usual 8. Intermediate results are calculated to 33 digits instead of the usual 20.

-Df	This option changes the way the compiler treats data items declared as COMP-1 and COMP-2. Some compilers use COMP-1 and COMP-2 to specify single- and double-precision floating-point data items. ACUCOBOL-GT, however, assigns a different meaning to COMP-1 and COMP-2 and uses FLOAT and DOUBLE to specify floating-point data items.When the `-Df` option is used, the compiler treats data items declared as COMP-1 as if they were declared FLOAT and data items declared as COMP-2 as if they were declared DOUBLE. With the `-Df` option, you have the following correspondence:
	`COMP-1   FLOAT    single precision`
	`COMP-2   DOUBLE   double precision`
	The `-Df` option makes it easier to compile code originally written for another compiler — one that used COMP-1 and COMP-2 to specify floating point data items. The `-Df` option lets you compile such code without having to change COMP-1 and COMP-2 to FLOAT and DOUBLE.

-Dl1/2/4/8	This option allows you to limit the maximum alignment modulus that will be used for SYNCHRONIZED data items. Normally, a synchronized data item is aligned on a 2-, 4-, or 8-byte boundary depending on its type. This option allows you to specify an upper bound to the modulus used. This is specified as a single digit that immediately follows the `-Dl` as part of the same argument. For example, `-Dl4` specifies that the maximum synchronization boundary is a 4-byte boundary. If you want to make programs that are compliant with the 88/Open COBOL specification, you should specify `-Dl4`.
-Dm	This option causes any data item whose underlying type is binary to be stored in the minimum number of bytes needed to hold it. Normally, binary types are stored in two, four, or eight bytes. This option allows storage in any number of bytes ranging from one to eight.
-Dq	Causes the QUOTE literal to be treated as an apostrophe, or single quotation mark, rather than as a double quotation mark ("). One exception to this is the HP e3000 TRANSFORM verb, in which QUOTE is always treated as a double quotation mark.

Option	Definition
-Ds	This causes USAGE DISPLAY numeric items with no SIGN clause to be treated as if they were described with the SIGN IS TRAILING SEPARATE clause. Several versions of RM/COBOL behave this way (all versions before 2.0, and some versions afterward).
-Dv	This option allows you to specify the default byte (initial value) used to initialize any data item not otherwise initialized when the program is loaded. The option must be followed by an equals sign (=) and the decimal value of the byte to use (for all current platforms, this is the ASCII value of the desired character). For example, to fill memory with the NULL character, use `-Dv=0`. To fill memory with the ASCII space character, use `-Dv=32`.
-Dw32	This option is checked for compatibility with the system's bit arrangement.
-Dw64	This option is checked for compatibility with the system's bit arrangement.

Truncation Options

The truncation options enable you to control the truncation of certain data items.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Dz	All binary and packed-decimal data types ignore their PICTURE when determining the largest value they can hold. The PICTURE is not used when moving to a nonnumeric destination (the largest possible value determines the number of digits moved instead).
-noTRUNC	All binary data types ignore their PICTURE when determining the largest value they can hold. However, the PICTURE is used when moving data from a binary number to a nonnumeric data item. The name of this option is similar to the name used by some other COBOL systems that behave this way.
-truncANSI	Full ANSI COBOL rules are in place. Each numeric data item stores values up to its PICTURE in size. A small number of USAGE types provide exceptions (such as COMP-X and COMP-5). Values larger than allowed by the PICTURE are truncated using the standard size rules when the data item is the target of a MOVE statement; however, COMP-5 items do use their PICTURE when moving a value to a nonnumeric data item. The results of an arithmetic overflow (without the SIZE phrase) are undefined.

Comments:

The -Dz truncation option is not supported in Managed COBOL.

Video Options

The video options enable you to control the behavior of certain items displayed to screen.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Vc	This option causes any ACCEPT statement that contains a numeric or numeric edited receiving field to be treated as if the CONVERT phrase were also specified.
-Vd	This option causes non-USAGE DISPLAY numeric items to be converted to USAGE DISPLAY before the screen display occurs. This option is always on.

Warning and Error Options

The warning and error options enable you to set the error threshold before a object file will stop executing.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-a	This flag is now obsolete and should not be used.
-Qm	This option specifies the number of errors the compiler reports before it exits. The option must be followed by a positive numeric argument, which is the maximum number of errors the compiler reports before it exits. The default value is `100`.

Debugging Options

The debugging options enable you to generate and execute object files suitable for debugging.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Note: The results of these options may differ slightly from how they appear in ACUCOBOL-GT COBOL, as they map to the debugging Compiler directives in Visual COBOL.

Option	Definition
-Ga, -Gd, -Gl, -Gs, -Gy	These options map to the ANIM Compiler directive setting.
-Gz	This option maps to the NOANIM Compiler directive setting.

Miscellaneous Options

The miscellaneous options enable you to control a number of aspects of the generated object files, such as bounds checking and optimization.

The following compiler options are supported in Visual COBOL when using `ccbl` from the command line or the ACUOPT Compiler directive.

Option	Definition
-Za	Causes the compiler to generate code to test array references at runtime. If an index is used which is out-of-bounds, the runtime system displays an error message showing the index value and the allowed bounds. (This causes some extra code to be generated and prevents certain table optimizations from occurring, so it should be turned off once a program is fully debugged.) With this option, the compiler does not re-use previously computed index values.
-Zc	This compact option optimizes for smaller code instead of faster code. This option is accepted by the compiler, but is ignored.
-Zd	Although still supported, this option has been replaced by the `-Gd` option. Both options produce the same results.
-Zg	Enables the use of segmentation (overlays) in the source. If this option is not used, section numbers will be ignored. This option is accepted by the compiler, but is ignored.
-Zi	Causes the program to be compiled as if it had the `IS INITIAL PROGRAM` phrase specified in its PROGRAM-ID paragraph.
-Zl	All data items may be larger than 64 KB. This option is obsolete.
-Zn	This turns off ACUCOBOL-GT's local optimizer. This is useful primarily to see if the optimizer is introducing errors in the generated object code. This option also prevents the compiler from re-using previously computed index values. This option is accepted by the compiler, but is ignored.
-Zs	Although still supported, this option has been replaced by the `-Gy` option. Both options produce the same results.
-Zy	This option lets you treat ACCEPT FROM DATE as ACCEPT FROM CENTURY-DATE, and ACCEPT FROM DAY as ACCEPT FROM CENTURY-DAY. If you use this option, the 4-digit year format will be used for ACCEPT FROM DATE providing that: • The receiving field is numeric or numeric edited and contains eight or more integer digits; or • The receiving field is not numeric or numeric edited and contains eight or more character positions. If neither of the above conditions applies, then ACCEPT FROM DATE will return its normal 6-digit format even if you use `-Zy`.
-Zr0	This option tells the compiler not to allow recursive PERFORMs. Event procedures require the ability to do recursive PERFORMs.

Option	Definition
-Zr1	This option tells the compiler to allow recursive PERFORMs. Event procedures require the ability to do recursive PERFORMs.

32- and 64-Bit Code Generation

When compiling with the DIALECT"ACU" directive, the Compiler generates intermediate code that is bit independent. By using `ccbl`, you can specify 32 or 64-bit intermediate code.

When compiling for generated code you must specify 32 or 64-bit. Visual COBOL is bit-specific and does not support cross-bit generation.

To produce bit-specific code, use `ccbl` from either a 32-bit or 64-bit command prompt.

ACUCOBOL-GT Conversion Issues

The syntax of most ACUCOBOL-GT source programs when submitted to run on this COBOL system will be accepted and run successfully. However, sometimes this COBOL system might reject some of the syntax in the original source program, or might cause your program to behave unexpectedly at run-time.

The following is not an exhaustive list of the restrictions of using ACUCOBOL-GT source code in Visual COBOL. In most cases, if your code includes ACUCOBOL-GT features not supported by Visual COBOL, you will receive a Compiler error.

Unsupported Features

The following features of ACUCOBOL-GT are not supported in Visual COBOL

- The ACUCOBOL-GT multi-threading model.
- The ACUCOBOL-GT Thin Client technology.
- The Graphical Technology (GT).

Configuration Files and Configuration Variables

ACUCOBOL-GT configuration files and configuration variables are not supported in Visual COBOL.

Visual COBOL uses different configuration files and variables. You need to review your existing ACUCOBOL-GT configuration to determine which settings are relevant for use and which settings have Visual COBOL equivalents.

For example, some configuration settings for handling Vision files can be set in the default File Handler in Visual COBOL.

Some ACU configuration variables are not necessary or applicable in Micro Focus COBOL (for example, PERFORM_STACK), and the functionality of others is covered by the Micro Focus compile and run-time options (for example, A_CHECKDIV).

Screen Descriptions

Visual COBOL and extend® differ in their support for some of the Screen Description phrases.

In Visual COBOL, the following phrases of the Screen Description entry are not supported and should be removed from your programs:

AFTER
BEFORE
EXCEPTION

Truncation Options in Managed Code

The -Dz truncation option is not supported in Managed COBOL.

It is, however, supported in native COBOL, using the ACUOPT Compiler directive.

Unable to use pipes in the SELECT statement

When using the Vision file handler, you cannot use pipes in the ASSIGN clause of the SELECT statement.

ACUFH does not currently support assigning files to pipes - for example:

```
select test-file assign to "-P %TMP% cmd /c dir *.* > %TMP%"
```

1. Create a subprogram which does not use a CALLFH"ACUFH" statement and which handles the required pipes. The syntax for assigning file to pipes is different in the Visual COBOL File Handler:

   ```
   select test-file assign to "<cmd /c dir *.*
   ```

   ```
   select test-file assign to "<ls *"
   ```
2. Call the subprogram from the program using CALLFH"ACUFH".

For more information, read *Programming* > *File Handling* > *File Handling Guide* > *Filenames* > *Setting Up Pipes* in the product Help.

Using Vision Indexed Files

This COBOL development system allows you to continue to use your existing ACUCOBOL-GT data files, including Vision indexed data files.

Alternatively, you can use the data migration tool, `ACU2MFDataMigration.exe`, to convert Vision data files to Micro Focus format. The tool is available from the folder in which the product samples are installed.

Configuring Access to Vision Files

To handle Vision files, you map a file to a compatible IDXFORMAT in the File Handler configuration file.

Within the configuration file, you can map an IDXFORMAT to all files in a particular folder, all files with a specific file extension, or a single file. See *Format of the Configuration File* for the tags that you can use for the mapping, and the order in which settings in these tags are applied.

The order that the mapping is applied is important, as conflicting settings can be overwritten; for example, the following excerpt of the configuration file sets all files in `c:\files\rmfiles` to IDXFORMAT 21 and all files with a `.DAT` extension to IDXFORMAT 17:

```
[FOLDER:C:\\files\\rmfiles]
IDXFORMAT=21

[*.DAT]
IDXFORMAT=17
```

If there is a `.DAT` file in `c:\files\rmfiles`, the mappings are applied according to the type of tag. In the case above, mappings in the extension tag are applied after mappings in the FOLDER tag, and so the `.DAT` file in that directory has an IDXFORMAT of 17.

By default, the File Handler handles all sequential and relative data files, but if you want to handle them through the Vision file handler, use the INTEROP=ACU configuration option; however, in cases where the INTEROP and IDXFORMAT mappings conflict, the INTEROP setting will override IDXFORMAT for your Vision indexed data files.

Vision Related Utilities

Vision provides a series of utilities that enable you to manipulate Vision files from the command line.

Each utility is available in a 32-bit and a 64-bit version, located in `%ProgramFiles(x86)%\Micro Focus\Visual COBOL\binn` and `\binn64` respectively.

Commands

vutil32	Rebuilds a file that has become corrupt, or one that contains a large number of deleted records that you want to remove from the file.
vio32	Enables you to collect a group of files together into archives, and allows you to extract some or all of these files from these archives.
logutil32	Enables you to examine and edit an ACUCOBOL-GT transaction log file.
acusort	Enables you to sort or merge Vision files.

ACUCOBOL-GT Library Routines

This COBOL development system provides a number of ACUCOBOL-GT library routines in native and managed code.

C$CALLEDBY

Returns the name of the caller of the currently running COBOL program or spaces if no caller exists or if the caller is unknown.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$CALLEDBY"
    USING CALLING-PROGRAM
    GIVING CALL-STATUS
```

Parameters

CALLING-PROGRAM PIC X(n)	Contains the name of the calling program or spaces if no caller exists or if the caller is unknown. The runtime will use as much space for the name or spaces as the COBOL program allows. If the object being called is in an object library, the program returns the PROGRAM-ID. If the object is not in an object library, the disk name is returned.
CALL-STATUS PIC S99	This parameter receives one of the following values: 1 - Routine called by another COBOL program 0 - Routine is the main program; no caller exists -1 - Caller unknown; routine not called by a COBOL program

Compatibility Issues

None.

C$CALLERR

Retrieves the reason why the last CALL statement failed. For accurate information, it must be called before any other CALL statement is executed.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$CALLERR"
    USING ERR-CODE, ERR-MESSAGE
```

Parameters

ERR-CODE PIC X(2)

This parameter receives one of the following values:

01	Program file missing or inaccessible
02	Called file not a COBOL program
03	Corrupted program file
04	Inadequate memory available to load program
05	Unsupported object code version number
06	Recursive CALL of a program
07	Too many external segments
08	Large-model program not supported (returned only by runtimes that do not support large-model programs)
09	Exit Windows and run "share.exe" to run multiple copies of "wrun32.exe" (returned only by Windows runtimes)
14	Japanese objects are not supported (returned only by runtimes that do not support Japanese objects)

ERR-MESSAGE PIC X(n) (optional)

This routine may optionally be passed a second alphanumeric parameter. This parameter is filled in with a descriptive message about the error encountered.

Compatibility Issues

- Only ERR-CODE 01 is returned in this COBOL system.
- ERR-MESSAGE is always set to SPACES.

C$CHDIR

Changes the current working directory.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$CHDIR"
    USING DIR-NAME, ERR-NUM
```

Parameters

DIR-NAME PIC X(n)	Contains the name of the new directory, or spaces. The "@[DISPLAY]:" for Thin Client support is allowed. For example: `C$CHDIR "@[DISPLAY]:C:\path"` In Thin Client environments, to get the current default directory on the display host, DIR_NAME should contain "@[DISPLAY]:" followed by spaces.
ERR-NUM PIC 9(9) COMP-4 (optional)	Holds the returned error number, or zero on success.

Comments

If a second USING parameter is passed, it must be described as PIC 9(9) COMP-4. This parameter will be set to ZERO if the directory change is successful. Otherwise, it will contain the operating system's error number.

If DIR-NAME contains spaces, then the current default directory is returned in it. In this case, ERR-NUM is not used. Otherwise, DIR-NAME should contain the name of a directory to make the new default directory. On Windows machines, this can include a drive letter. If you pass ERR-NUM, it will be set to zero if the change was successful. Otherwise, ERR-NUM will contain the error value returned by the operating system.

On some systems (such as VMS), it is legal to switch to a directory that does not exist, while other systems (Windows, UNIX) do not allow it.

The behavior of this routine is affected by the FILENAME_SPACES configuration variable. The value of FILENAME_SPACES determines whether spaces are allowed in a file name.

IMPORTANT

If you use C$CHDIR, create a CODE_PREFIX configuration entry to locate your object files. Ensure that all of the search locations specified by the CODE-PREFIX are full path names. Do not use the current directory or any relative path names in the CODE_PREFIX. Without a full path name, the runtime system may be unable to find your object files if it needs to re-open them.

For example, the runtime system must occasionally re-open an object file when:

- you are using the source debugger
- the program contains segmentation (overlays)
- you are using object libraries

If the object file was initially found in the current directory or a directory specified relative to the current directory, and you then change the current directory with the C$CHDIR routine, the runtime system will not be able to find the object file if it needs to re-open it. This will cause a fatal error and your program will halt.

If you use C$CHDIR and you are running in debug mode, be sure to set CODE_PREFIX in the configuration file, not in the environment. You may set CODE_PREFIX in the environment when you are not in debug mode.

Compatibility Issues

- "@[DISPLAY]" is not supported in this COBOL system.
- The FILENAME_SPACES configuration variable is not supported in this COBOL system. To use filenames that contain spaces, enclose them in quotation marks.
- The CODE_PREFIX configuration variable is not supported in this COBOL system.

C$COPY

Creates a copy of an existing file.

Syntax:

```
CALL "C$COPY"
   USING source-file, dest-file, [file-type,]
   [GIVING status]
```

Parameters:

source-file

PIC X(n)

dest-file

PIC X(n)

file-type

PIC X

status

Any numeric type

On Entry:
source-file

The path name of the file to be copied

dest-file

The path name of the destination file

file-type

The file organization of the source file.It must be one of: S (for sequential), R (for relative) or I (for indexed).

This defaults to S if not specified.

On Exit:
copy-status

Returns zero if the copy is successful, or non-zero if not.

Comments:

To obtain an extended file status code for this operation, define `status` as comp xx comp-x and follow the example in *Extended File Status Codes*.

C$DELETE

Deletes a file.

Syntax:

```
CALL "C$DELETE"
   USING file-name, [file-type,]
   [GIVING status]
```

Parameters:
file-name

PIC X(n)

file-type

PIC X

status

Any numeric type

On Entry:
file-name

The pathname of the file to be deleted

file-type

The file organization of the filename. It must be one of: S (for sequential), R (for relative) or I (for indexed).

This defaults to S if not specified.

On Exit:
status

Returns zero if the delete is successful, or non-zero if not.

Comments:

To obtain an extended file status code for this operation, define `status` as comp xx comp-x and follow the example in *Extended File Status Codes*.

C$FILEINFO

Retrieves some operating system information about a given file.

Syntax:

```
CALL "C$FILEINFO"
   USING file-name, file-info
   GIVING status
```

Parameters:
file-name

PIC X(n)

file-info

Define the following group

```
01 file-info
  03 file-size  pic x(8) comp-x.
  03 file-date  pic 9(8) comp-x.
  03 file-time  pic 9(8) comp-x.
```

status

Any numeric type

On Entry:
file-name

The name of the file

On Exit:
file-info

The group item to receive the file information

status

Returns zero if the delete is successful, or non-zero if not.

Comments:

To obtain an extended file status code for this operation, define `status` as comp xx comp-x and follow the example in *Extended File Status Codes*.

C$GetLastFileOp

Retrieves the last COBOL I/O operation performed.

Use this library routine within a declarative procedure after an I/O error has occurred.

Syntax:

```
CALL "C$GetLastFileOp" USING operation
```

Parameters:
operation

PIC X(20)

On Exit:

operation The name of the last I/O operation performed. The valid operations returned are:

Close	ReadPreviousLock
Commit	ReadPreviousNoLock
Delete	Rewrite
DeleteFile	Rollback
Open	Start
ReadLock	StartTransaction
ReadNextLock	Unlock
ReadNextNoLock	UnlockAll
ReadNoLock	Write

Comments:

If the operation is longer than 20 characters, it is truncated to the right.

If the value SPACES is returned that indicates that no operation is available.

C$LIST-DIRECTORY

The C$LIST-DIRECTORY routine lists the contents of a selected directory. Each operating system has a unique method for performing this task. C$LIST-DIRECTORY provides a single method that will work for all operating systems.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$LIST-DIRECTORY"
    USING OP-CODE, parameters
```

Parameters

OP-CODE PIC 99 COMP-X	Indicates which C$LIST-DIRECTORY operation to perform. The operations are described below.

Parameters vary depending on the op-code chosen.

Parameters provide information and hold results for the op-code specified. These parameters are described below.

Description

C$LIST-DIRECTORY allows you to get the names of files residing in a given directory. It accomplishes this through three distinct operations. The first operation opens the specified directory. The second operation returns the filenames in the list, one-at-a-time. The third operation closes the directory and deallocates all memory used by the routine. C$LIST-DIRECTORY has the following operation codes (defined in `acucobol.def`):

LISTDIR-OPEN (VALUE 1)

Opens the specified directory. It has two parameters:

Directoryname PIC X(n)

Contains the name of the directory to open. This directory must exist, and you must have permissions to read the directory. You may use remote name syntax if AcuServer is installed on the remote machine. The "@[DISPLAY]:" for Thin Client support may be used. For example:

```
C$LIST-DIRECTORY using listdir-open,
"@[DISPLAY]:C:\path", pattern
```

Pattern PIC X(n)

Specifies the type of filename for which to search. This routine supports "wildcards," meaning that the character "*" will match any number of characters, and the character "?" will match any single character. For example, you can search by file suffix (*.def) or by a common part of a file name (acu*).

If the call to LISTDIR-OPEN is successful, RETURN-CODE contains a handle to the list. The value in RETURN-CODE should be moved to a data item that is USAGE HANDLE. That data item should be passed as the directory handle to the other C$LISTDIRECTORY operations. If the call to LISTDIR-OPEN fails (if the directory does not exist, contains no files, or you do not have permission to read the directory), RETURN-CODE is set to a NULL handle.

LISTDIR-NEXT (VALUE 2)

Reads each filename from the open directory. It has two parameters:

Handle USAGE HANDLE

The handle returned in the LISTDIR-OPEN operation.

Filename PIC X(n)

The location of the next filename to be returned. If the directory listing is finished, it is filled with spaces.

The call to LISTDIR-NEXT can include an additional argument, LISTDIR-FILE-INFORMATION (defined in "acucobol.def"), which receives information about the returned file name. This is an optional group item which returns information about the following data items:

LISTDIR-FILE-TYPE

The file type can be one of the following:

B = block device
C = character device
D = directory
F = regular file
P = pipe (FIFO)
S = socket
U = unknown

LISTDIR-FILE-CREATION-TIME

The creation time is the date (and time) that the file was originally created.

LISTDIR-FILE-LAST-ACCESS-TIME

The last access time is the date (and time) that the file was last accessed by some application (usually when the file was queried in some way).

LISTDIR-FILE-LAST-MODIFICATION-TIME

The last modification time is the date (and time) the file was last written to.

LISTDIR-FILE-SIZE

The size of the file is given in bytes.

LISTDIR-CLOSE (VALUE 3)

Releases the resources used by the other operations. It must be called to avoid memory leaks. It has one parameter, handle, which is the same data item used by the LISTDIR-NEXT operation.

Handle USAGE HANDLE

The handle returned in the LISTDIR-OPEN operation.

Note: Because the supported file types vary by operating system, The data items in the above list have slightly different meanings depending on your operating system. Even on operating systems that support these values, some file systems may not. Some versions of the UNIX® operating system may change these values when permissions are changed. Refer to your operating system documentation for specific definitions.

Example

The following example lists the contents of a directory with repeated calls C$LISTDIRECTORY:

```
WORKING-STORAGE SECTION.
copy "def/acucobol.def".
01  pattern       pic x(5) value "*.vbs".
01  directory     pic x(20) value "/virusscan".
01  filename      pic x(128).
01  mydir         usage handle.
PROCEDURE DIVISION.
MAIN.
* CALL LISTDIR-OPEN to get a directory handle.
    call "C$LIST-DIRECTORY"
       using listdir-open, directory, pattern.
    move return-code to mydir.
    if mydir = 0
       stop run
    end-if.
* CALL LISTDIR-NEXT to get the names of the files.
* Repeat this operation until a filename containing only
* spaces is returned.  The filenames are not necessarily
* returned in any particular order.  Filenames may be
* sorted on some machines and not on others.
    perform with test after until filename = spaces
       call "C$LIST-DIRECTORY"
          using listdir-next, mydir, filename
    end-perform.
* CALL LISTDIR-CLOSE to close the directory and deallocate
* memory. Omitting this call will result in memory leaks.
    call "C$LIST-DIRECTORY" using listdir-close, mydir.
    stop run.
```

Compatibility Issues

- This routine is not supported in managed COBOL.
- You must compile with the DIALECT"ACU" Compiler directive when using this library routine.
- "@[DISPLAY]" is not supported in this COBOL system.

C$MAKEDIR

Creates a new directory.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

C$MAKEDIR can make a directory only one level lower than an existing directory and cannot create more than one level at a time.

Usage

```
CALL "C$MAKEDIR"
    USING DIR-NAME GIVING STATUS-CODE
```

Parameters

DIR-NAME PIC X(n)	Contains the name of the directory to be created. This should be either a full path name or a name relative to the current directory. You may use remote name syntax in combination with AcuServer to create a directory on a remote machine. The "@[DISPLAY]:" annotation for Thin Client support may also be specified. For example: `C$MAKEDIR "@[DISPLAY]:C:\path"`
STATUS-CODE Numeric data item.	Receives the return status of the call to create a directory. A return status of zero indicates that the directory was successfully created; a status of one ("1") indicates otherwise. The behavior of this routine is affected by the FILENAME_SPACES configuration variable. The value of FILENAME_SPACES determines whether spaces are allowed in a file name.

Compatibility Issues

- "@[DISPLAY]" is not supported in this COBOL system.
- The FILENAME_SPACES configuration variable is not supported in this COBOL system. To use filenames that contain spaces, enclose them in quotation marks.

C$MEMCPY (Dynamic Memory Routine)

Copies bytes between any two memory locations.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$MEMCPY"
    USING, BY VALUE, DEST-PTR, SRC-PTR, NUM-BYTES
```

Parameters

DEST-PTR USAGE POINTER or USING BY REFERENCE	Contains the address of the first byte of the destination.
SRC-PTR USAGE POINTER or USING BY REFERENCE	Contains the address of the first byte of the source.
NUM-BYTES USAGE UNSIGNED-INT or an unsigned numeric literal	Indicates the number of bytes to copy.

Description

This routine copies NUM-BYTES bytes of memory from the address contained in SRC-PTR to the address contained in DEST-PTR. This routine is functionally similar to the M$COPY (Dynamic Memory Routine) routine except that parameters are passed by value instead of by reference. This routine can be used in

cases where M$PUT and M$GET are not adequate. Note that this routine is relatively dangerous to use. It does not perform any error checking and can easily cause memory access violations if you pass it incorrect data. In other words, this routine is a very low-level routine and should be used cautiously.

You do not need to pass POINTER data items for SRC-PTR and DEST-PTR. If you prefer, either or both can be replaced by a data item passed BY REFERENCE. If you do this, then the address of the data item is passed to C$MEMCPY. For example, you can copy 10 bytes to DEST-ITEM from the memory address contained in SRC-PTR with:

```
     CALL "C$MEMCPY"
        USING BY REFERENCE DEST-ITEM, BY VALUE SRC-PTR, 10
```

Compatibility Issues

None.

C$MYFILE

Returns the filename of the disk file containing the currently executing program.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

This is especially useful if the disk file is an object library.

Usage

```
CALL "C$MYFILE"
    USING PROGRAM-NAME
    GIVING CALL-STATUS
```

Parameters

PROGRAM-NAME PIC X(n)	Indicates the name of the disk file containing the currently executing program, if known. The runtime will use as much space for the name of the file as the COBOL program allows. This parameter will contain the filename just as the runtime received it. For example, if an object library is loaded as `../ardir/myarlib.lib`, and a program in `myarlib.lib` calls this routine, PROGRAM-NAME will have a value of `../ardir/myarlib.lib`.
CALL-STATUS PIC S99.	This parameter receives one of the following values: 1 - PROGRAM-NAME was filled successfully -1 - Program name unknown

Compatibility Issues

None.

C$NARG

This routine returns the number of parameters passed to the current program.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$NARG"
    USING NUM-PARAM
```

Parameter

NUM-PARAM COMP-1

Description

This routine must be called with one USING parameter that must be a COMP-1 data item. This data item is filled in with the number of parameters. If the calling program is a subprogram, then this will be the number of USING items in the CALL statement that initiated the program. If the calling program is a main program, then this will be the number of CHAINING parameters passed from the runcbl command line or the CHAIN statement that initiated the program. C$NARG works only when the program is a called subroutine. It does not work with the "CALL RUN" form of the CALL verb.

Compatibility Issues

- This routine is not supported in managed COBOL.
- Set the Compiler directive COMP1(BINARY) to set ACUCOBOL-GT behavior for COMP-1 data items.
- The "CALL RUN" statement is not supported in this COBOL system.
- In ACUCOBOL-GT COBOL, the number of parameters passed is calculated by the number of parameters specified in the USING phase of the CALL statement in the calling program. In this COBOL system, the number of parameters passed is calculated by the number of parameters the calling program actually receives.

C$PARAMSIZE

This routine returns the number of bytes actually passed by the caller for a particular parameter.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "C$PARAMSIZE"
    USING PARAM-NUM,
    GIVING PARAM-SIZE
```

Parameters

PARAM-NUM numeric parameter	This value is the ordinal position in the Procedure Division's USING phrase of the parameter whose size you want to know.
PARAM-SIZE any numeric data item	This item receives the number of bytes in the data item actually passed by the caller.

Description

This routine returns the actual size (in bytes) of a data item passed to the current program by its caller. You pass the number (starting with `1`) of the data item in the Procedure Division's USING phrase, and C $PARAMSIZE will return the size of the corresponding item that was actually passed. This can be useful for handling data items of unknown size.

For example, suppose that you wanted to write a routine that could convert any data item to upper-case, up to 10000 bytes in size. This routine could look like this:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAKE-UPPERCASE.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  PARAM-SIZE    PIC 9(5).

LINKAGE SECTION.
77  PASSED-ITEM   PIC X(10000).

PROCEDURE DIVISION USING PASSED-ITEM.
MAIN-LOGIC.
   CALL "C$PARAMSIZE" USING 1, GIVING PARAM-SIZE
   INSPECT PASSED-ITEM( 1 : PARAM-SIZE )
      CONVERTING "abcdefghijklmnopqrstuvwxyz"
      TO "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  EXIT PROGRAM.
```

In this example, if you do not use C$PARAMSIZE, you have to pass a full 10000 bytes to this routine or you get a memory usage error. By using C$PARAMSIZE and reference modification, only the memory actually passed is referenced, and there is no error. C$PARAMSIZE works only when the program is a called subroutine. It does not work with the "CALL RUN" form of the CALL verb.

If you pass a subitem of a linkage item in a CALL statement and the subprogram calls C$PARAMSIZE with requesting the size of the parameter, it will get the size as described in the linkage section of the calling program, unless that subitem is the first item of the linkage item. In that case, the size returned will be the size of the original item.

Compatibility Issues

- This routine is not supported in managed COBOL.
- In this COBOL system, the size of the item as specified in the calling program is always returned.

C$RERR

Returns extended file status information for the last I/O statement.

Syntax:

```
CALL "C$RERR" USING extend-stat [text-message, status-type]
```

Parameters:

extend-stat

PIC X(5) or larger

text-message

PIC X(n)

Note: This optional parameter is ignored in this COBOL system.

status-type

PIC 9

Note: This optional parameter is ignored in this COBOL system.

On Exit:

extend-stat	Returns the extended file status caused by the last file I/O

Comments:

The statuses returned are listed in the file status table found in *Appendix E* of the ACUCOBOL-GT product documentation. If the file status (first two characters) is 30, the remainder of the information is the

operating system's status code explaining what caused the error. On some systems, the operating system requires more than two digits for its status codes. That is why the C$RERR routine may be passed a field that is larger than four characters.

Whenever an error 30 occurs, the operating system's status value is returned in this extended field. The number returned is a left-justified decimal value. If the receiving field is too small, the right-most digits are returned. If the receiving field is too large, the excess characters are filled with spaces.

C$RERRNAME

Returns the name of the last file used in an I/O statement.

Use it in conjunction with C$RERR to diagnose file errors.

Syntax:

```
CALL "C$RERRNAME" USING file-name
```

Parameters:

file-name

PIC X(n)

On Exit:

file-name The name of the last file that was involved in an I/O statement.

Note: The filename is the one specified in the ASSIGN clause.

M$ALLOC (Dynamic Memory Routine)

Allocates a new area of dynamic memory.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$ALLOC"
    USING ITEM-SIZE, MEM-ADDRESS
```

Parameters

ITEM-SIZE Numeric parameter	This indicates the number of bytes to allocate. This must be greater than zero.
MEM-ADDRESS USAGE POINTER	This holds the return value, either the address of the allocated memory or NULL if the allocation fails.

Comments

The maximum amount of memory you may allocate in one call depends on the host machine, but is at least 65260 bytes for all machines (providing that much memory is available). M$ALLOC adds some overhead to each memory block allocated. This ranges between 4 and 16 bytes depending on the machine architecture. Also, each operating system will typically add its own overhead. The debugger's `U` command reports the amount of memory you have currently allocated via M$ALLOC. The overhead added by M $ALLOC is included in the total shown, but the operating system's overhead is not. Memory allocated by M $ALLOC is initialized to binary zeros (LOW VALUES).

If you try to allocate more memory than the environment can give you, M$ALLOC will return NULL, and no memory will be allocated.

Compatibility Issues

None.

M$COPY (Dynamic Memory Routine)

Copies a region of memory from one location to another.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$COPY"
    USING DEST-PTR, SRC-PTR, NUM-BYTES
```

Parameters

DEST-PTR USAGE POINTER	Contains the address of the first byte of the destination region.
SRC-PTR USAGE POINTER	Contains the address of the first byte of the source region.
NUM-BYTES Numeric parameter	Indicates the size of the memory region to be copied.

Description

This routine copies NUM-BYTES from the address contained in SRC-PTR to the address contained in DEST-PTR. Note that this routine is relatively dangerous to use. No boundary checking is performed to ensure that the address range is valid, so memory access violations may result if you pass it incorrect data.

This routine is functionally similar to the C$MEMCOPY routine except that parameters are passed by reference instead of by value. For example, you can copy 10 bytes to DEST-PTR from the memory address contained in SRC-PTR with:

```
     CALL "M$COPY"
        USING DEST-PTR, SRC-PTR, 10
```

Compatibility Issues

None.

M$FILL (Dynamic Memory Routine)

Sets a region of memory to a constant value.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$FILL"
    USING DEST-PTR, BYTE-VALUE, NUM-BYTES
```

Parameters

DEST-PTR USAGE POINTER	Contains the address of the first byte of the region to be filled.
BYTE-VALUE Alpha-numeric parameter	Contains the value with which to fill the memory region.
NUM-BYTES Numeric parameter	Indicates the size of the memory region.

Description

This routine fills NUM-BYTES with BYTE-VALUE starting at address DEST-PTR. The parameters are passed BY REFERENCE. This routine does not do any boundary checking to make sure that the address range is valid.

Compatibility Issues

None.

M$FREE (Dynamic Memory Routine)

Frees a previously allocated piece of memory.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$FREE"
    USING MEM-ADDRESS
```

Parameter

MEM-ADDRESS USAGE POINTER	Must point to a memory area previously allocated by M $ALLOC.

Comments

Use M$FREE to release a memory block allocated by M$ALLOC. This memory is returned to the pool of memory available for use by the runtime. On most operating systems, this memory is still associated with the runtime's process, so it cannot be used by any other processes. On a few systems, this memory may be made available to the operating system for re-use by other processes.

It is an error to attempt to use a block of memory once it has been freed. It is also an error to free a block of memory more than once or to free a memory address that has never been allocated. Any of these errors can lead to memory access violations. The runtime attempts to detect these errors and avoid them, but it cannot detect all such errors.

Compatibility Issues

None.

M$GET (Dynamic Memory Routine)

Retrieves data from an allocated memory block.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$GET"
    USING MEM-ADDRESS, DATA-ITEM, DATA-SIZE, DATA-OFFSET
```

Parameters

MEM-ADDRESS USAGE POINTER	Must point to a memory area previously allocated by M $ALLOC.
DATA-ITEM Any data item	Data from the memory block will be stored in this item.

DATA-SIZE Numeric parameter (optional)	The number of bytes to move from the memory block. If omitted, then the number of bytes is set to the size of the memory block (excluding overhead bytes).
DATA-OFFSET Numeric parameter (optional)	The location within the memory block from which to start the move. The first location is position `1`. If omitted, this value defaults to `1`.

Description

This routine retrieves data from the memory block at MEM-ADDRESS and stores it in DATA-ITEM. Regardless of the value of DATA-SIZE, no bytes are copied from past the end of the memory block. Note that the size of DATA-ITEM is not checked.

Compatibility Issues

None.

M$PUT (Dynamic Memory Routine)

Stores data in an allocated memory block.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

Usage

```
CALL "M$PUT"
    USING MEM-ADDRESS, DATA-ITEM, DATA-SIZE, DATA-OFFSET
```

Parameters

MEM-ADDRESS USAGE POINTER	Must point to a memory area previously allocated by M $ALLOC.
DATA-ITEM Any data item	This is the data that will be stored in the memory block.
DATA-SIZE Numeric parameter (optional)	The number of bytes to move to the memory block. If omitted, then the number of bytes is set to the size of the memory block (excluding overhead bytes).
DATA-OFFSET PIC 9(n), USAGE DISPLAY or COMP-4 (optional)	The location within the memory block from which to start the move. The first location is position `1`. If omitted, this value defaults to `1`.

Description

This routine copies DATA-ITEM into the memory pointed to by MEM-ADDRESS for DATA-SIZE bytes. Regardless of the value of DATA-SIZE, no bytes are copied that exceed the size of the memory block at MEM-ADDRESS.

Compatibility Issues

None.

RENAME

Renames a file.

Syntax:

```
CALL "RENAME"
   USING source-file, dest-file, [status,] [file-type]
```

Parameters:

source-file

PIC X(n)

dest-file

PIC X(n)

status

Any numeric type

file-type

PIC X

On Entry:

source-file

The path name of the file to be copied

dest-file

The path name of the destination file

file-type

The file organization of the source file. It must be one of: S (for sequential), R (for relative) or I (for indexed).

This defaults to S if not specified.

On Exit:

status

Returns zero if the rename is successful, or non-zero if not.

Comments:

To obtain an extended file status code for this operation, define `status` as comp xx comp-x and follow the example in *Extended File Status Codes*.

WIN$VERSION

Returns version information for Windows and Windows NT host platforms.

Note: This ACUCOBOL-GT library routine is available in this COBOL version. Any compatibility issues in this COBOL system are in the Compatibility Issues section at the end of the topic.

This routine provides more information about the system than is returned by the ACCEPT FROM SYSTEM-INFO statement.

Usage

```
CALL "WIN$VERSION"
    USING WINVERSION-DATA
```

Parameters

WINVERSION-DATA Group item as follows:	`01   WINVERSION-DATA.` `     03   WIN-MAJOR-VERSION      PIC X`

```
COMP-X.
    03  WIN-MINOR-VERSION     PIC X
COMP-X.
    03  WIN-PLATFORM          PIC X
COMP-X.
        88  PLATFORM-WIN-31   VALUE
1.
        88  PLATFORM-WIN-95   VALUE
2.
        88  PLATFORM-WIN-9X   VALUE
2.
        88  PLATFORM-WIN-NT   VALUE
3.
    03  WIN-WORDSIZE          PIC X
COMP-X.
        88  WIN-WORDSIZE-16   VALUE
1.
        88  WIN-WORDSIZE-32   VALUE
2.
        88  WIN-WORDSIZE-64   VALUE
3.
    03  WIN-
BUILDNUMBER                 PIC X(4)
COMP-X.
    03  WIN-
CSDVERSION                  PIC
X(128).
    03  WIN-SERVICEPACK-
MAJOR           PIC X COMP-X.
    03  WIN-SERVICEPACK-
MINOR           PIC X COMP-X.
    03  WIN-
SUITEMASK                   PIC X(4)
COMP-X.
    03  WIN-
PRODUCTTYPE                 PIC X
COMP-X.
        88 WIN-NT-
WORKSTATION           VALUE 1.
        88 WIN-NT-DOMAIN-
CONTROLLER     VALUE 2.
        88 WIN-NT-
SERVER                VALUE 3.
```

WINVERSION-DATA is found in the COPY library `winvers.def`.

Comments

Upon return from WIN$VERSION, all of the data elements contained in WINVERSION-DATA are filled in. If you call WIN$VERSION and the host machine is not a Windows or Windows NT system, the fields are set to zero.

The WINVERSION-DATA fields have the following meaning:

- `WIN-MAJOR-VERSION` - The major version number reported by Windows. See table below for possible values.
- `WIN-MINOR-VERSION` - The minor version number reported by Windows. See table below for possible values.

Windows Version	WIN-MAJOR-VERSION	WIN-MINOR-VERSION	Other
Windows 98	4	10	
Windows ME	4	90	
Windows XP	5	1	Not applicable.
Windows XP Professional x64 Edition	5	2	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION) && (SYSTEM_INFO.wProcessorArchitecture==PROCESSOR_ARCHITECTURE_AMD64
Windows NT	4	0	
Windows 2000	5	0	Not applicable
Windows Vista	6	0	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows 7	6	1	OSVERSIONINFOEX.wProductType == VER_NT_WORKSTATION
Windows Server 2008 R2	6	1	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Server 2008	6	0	OSVERSIONINFOEX.wProductType != VER_NT_WORKSTATION
Windows Server 2003 R2	5	2	GetSystemMetrics(SM_SERVERR2) != 0
Windows Home Sever	5	2	OSVERSIONINFOEX.wSuiteMask & VER_SUITE_WH_SERVER
Windows Server 2003	5	2	GetSystemMetrics(SM_SERVERR2) == 0

- `WIN-PLATFORM` - Provides a general description of the host system. If the host is Windows NT/ Windows 2000, the value is set to PLATFORM-WIN-NT. If the host is Windows 98, the value is set to PLATFORM-WIN-9X.
- `WIN-WORDSIZE` - This item is set to WIN-WORDSIZE-32 for a 32-bit runtime.
- `WIN-BUILDNUMBER` - Identifies the build number of the operating system.
- `WIN-CSDVERSION` - Indicates the latest Service Pack installed on the system. If no Service Pack has been installed the string is empty.
- `WIN-SERVICEPACK-MAJOR` - Indicates the major version number of the latest Service Pack installed on the system. If no Service Pack has been installed the value is 0.
- `WIN-SERVICEPACK-MINOR` - Indicates the minor version number of the latest Service Pack installed on the system. If no Service Pack has been installed the value is 0.
- `WIN-SUITEMASK` - This is a bit mask that identifies the product suites available on the system. Refer to the operating system documentation for a list of possible values.

- `WIN-PRODUCTTTYPE` - Identifies additional information about the system.

Compatibility Issues

The copybook `winvers.def` is not available in this COBOL system.

The following fields are not supported in this COBOL system:

- `WIN-BUILDNUMBER`
- `WIN-CSDVERSION`
- `WIN-SERVICEPACK-MAJOR`
- `WIN-SERVICEPACK-MINOR`
- `WIN-SUITEMASK`
- `WIN-PRODUCTTTYPE`

These fields will return spaces or zeroes, as appropriate.

ACUCOBOL-GT Windowing Syntax

Your COBOL system provides some support for ACUCOBOL-GT windowing syntax that enables you to draw lines and boxes on the terminal screen, and create virtual terminal windows on a physical terminal. All ACCEPT and DISPLAY statements then act within the current window (except for ACCEPT format 1, 2, or 3 statements, DISPLAY format 1 statements, and DISPLAY WINDOW/LINE/BOX statements). The syntax also enables underlying displays to be kept and restored.

Note: This functionality is supported in native COBOL only.

Windowing Syntax Summary

Your COBOL system includes the following syntax to support windowing:

- BEFORE TIME phrase in ACCEPT statement

  Format 5 of the ACCEPT statement has the BEFORE TIME phrase, which enables you to specify a timeout period. If the user does not enter data during this period, the statement is terminated automatically.
- DISPLAY WINDOW

  DISPLAY WINDOW creates a terminal window (a rectangular region of the screen) and makes it the current window. This is like a virtual terminal, in which screen positions used by subsequent ACCEPT/ DISPLAY statements are relative to the top left corner of the window.
- DISPLAY LINE

  DISPLAY LINE enables you to draw lines on the terminal (real or virtual). The best mode available on the terminal is used automatically. Used with the DISPLAY BOX statement, the DISPLAY LINE statement enables you to draw forms on the terminal.
- DISPLAY BOX

  DISPLAY BOX enables you to draw boxes on the terminal. The best mode available on the terminal is used automatically. Used with the DISPLAY LINE statement, the DISPLAY BOX statement enables you to draw forms on the terminal.
- CLOSE WINDOW

  CLOSE WINDOW removes a window. If you specify the window as being a POP-UP window, the underlying display can be restored.

Enabling Windowing Support

In order to use the windowing syntax, you must use the PREPROCESS"window1" Compiler directive.

You can specify this directive in one of two ways.

- In your source file, use the following line:

```
$SET preprocess"window1"
```

- From the command line, include the PREPROCESS"window1" directive:

```
cobol prog.cbl preprocess"window1" color endp;
```

The PREPROCESS "window1" directive must be the last Compiler directive apart from NOERRQ, AUTOCLOSE or COLOR. If an error is encountered, the Compiler asks if you wish to continue, and waits for your response. In order to disable this function, you must specify the NOERRQ directive after PREPROCESS"window1".

Windowing Support Syntax

The following sections give details of the windowing syntax enabled by the PREPROCESS"window1" directive.

The ACCEPT Statement

```
BEFORE TIME time-out
```

General Rules:

1. The BEFORE TIME phrase allows you to automatically terminate an ACCEPT statement after a certain amount of time has passed. The timeout value specifies the time to wait in hundredths of a second. For example, "BEFORE TIME 500" specifies a timer value of 5 seconds.
2. The user must enter data to the ACCEPT statement before the timer elapses. As soon as the user starts entering data, the timer is canceled and the user may take as much time as desired to complete the entry. If the user does not enter any data before the timer elapses, then the ACCEPT statement terminates.

The CLOSE WINDOW Statement

Format:

```
CLOSE WINDOW window-save-area
```

Syntax Rules:

1. *window-save-area* must be an elementary data item described with a PIC X(10) clause. It must have been the object of a POP-UP AREA phrase in a DISPLAY WINDOW statement.

General Rules:

1. The CLOSE WINDOW statement is used to remove popup windows created by the POP-UP AREA option of the DISPLAY WINDOW statement.
2. *window-save-area* must have been the object of a POP-UP phrase of a DISPLAY WINDOW statement that has been executed in this run unit. Furthermore, since that execution, it must not have been the object of a CLOSE WINDOW statement, nor can it have been modified by any other statement. Violation of these rules causes undefined results.
3. The CLOSE WINDOW statement restores the contents of the terminal screen that was in the active window when the corresponding DISPLAY WINDOW statement executed. In other words, the window that was created by that DISPLAY WINDOW statement is removed from the screen and replaced by the contents of the screen which were under that popup window.
4. The window that was active when the corresponding DISPLAY WINDOW statement executed becomes the active window, thereby becoming the top window and overlaying any other windows that might be present.

Comments:

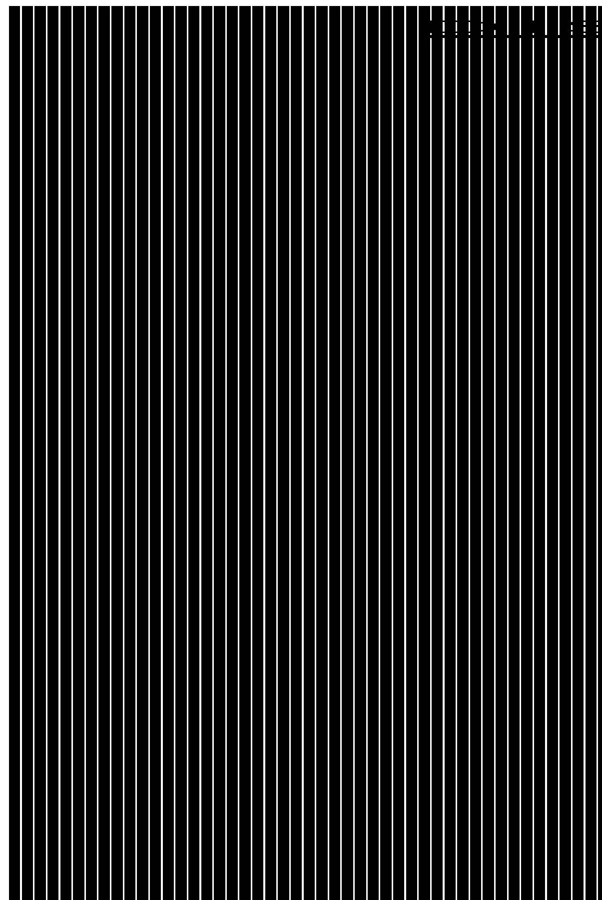
The current window is selected by closing windows identified by their respective *window-save-area* data items, as in the following example:

If five popup windows are created, *a*, *b*, *c*, *d* and *e* in that order:

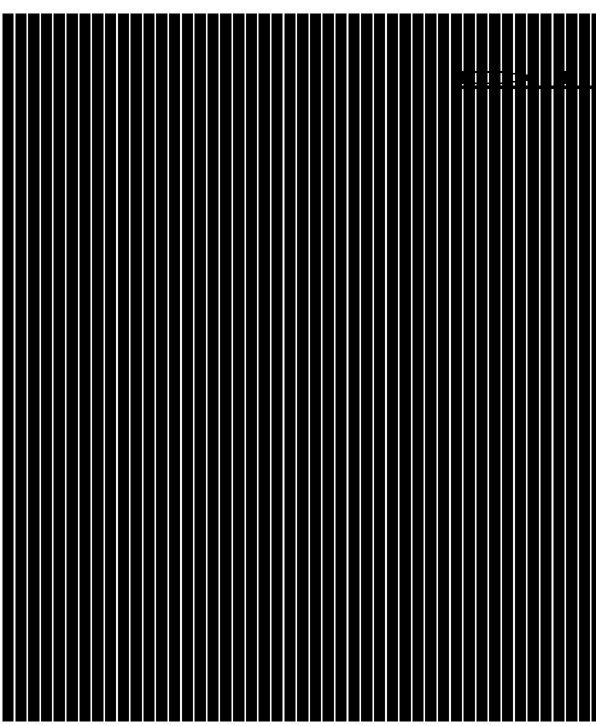
- If *d* is closed, *c* becomes current.
- If *b* is then closed, *a* becomes current.
- If *e* is subsequently closed, *c* becomes current again.

The DISPLAY Statement

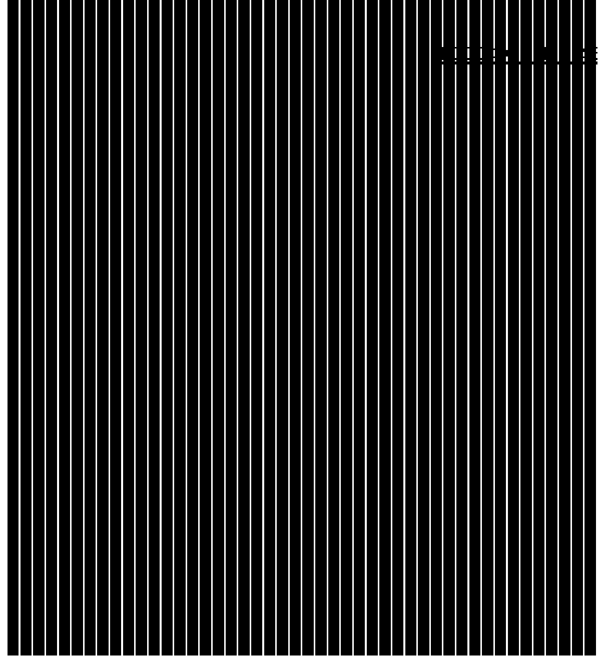
Format: for Format 1



Format: for Format 2



Format: for Format 3



Syntax Rules:

1. *line-num* is a numeric literal or data item that specifies the line position on the terminal screen. It must be a non-negative integer.
2. *col-num* is a numeric literal or data item that specifies the column position on the terminal screen. It must be a non-negative integer.

3. *length* is a numeric literal or data item that specifies the window-width, line-width or box-width in character positions. It must be a non-negative integer.
4. *height* is a numeric literal or data item that specifies the number of lines in the window, line or box. It must be a non-negative integer.
5. *title* is a non-numeric literal or alphanumeric data item.
6. *save-area* is an elementary data item described by a PIC X(10) clause.
7. COLUMN and COL are equivalent.
8. REVERSE and REVERSED and REVERSE-VIDEO are equivalent.
9. The COLOR phrase is supported only when the preprocessor directive COLOR is used. This adds support for existing non-Micro Focus syntax.
10. Exactly one of the SIZE or LINES phrases must be specified for a Format 2 DISPLAY statement.
11. *identifier-1*, *identifier-2*, *integer-1* and *integer-2* must take a value in the range 0 through 7 as follows:

0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown or yellow
7	white

12. *identifier-3* and *integer-3* must take a value which is obtained by adding together the appropriate values from the following:

Color	Foreground	Background
Black	1	32
Blue	2	64
Green	3	96
Cyan	4	128
Red	5	160
Magenta	6	192
Brown	7	224
White	8	25

Note:

The foreground color values for use with the COLOR phrase are different from the standard Micro Focus color values for FOREGROUND-COLOR, BACKGROUND-COLOR and so on.

In addition you can specify the following video attributes with the following values:

Reverse video	1024
Low intensity	2048
High intensity	4096
Underline	8192

Blink	16384

13. If a COLOR phrase is present at the same time as FOREGROUND-COLOR and/or BACKGROUND-COLOR, then the colors defined in the COLOR phrase are ignored, but any non-color attributes are actioned (where appropriate).

General Rules: for All Formats

1. The LINE and COLUMN phrases must specify a line or column on the physical screen.
2. On color systems, both the settings given in COLOR or FOREGROUND-COLOR and BACKGROUND-COLOR and other attribute settings (for example blink) are used. On monochrome systems, all color information supplied is ignored, and only other attribute settings used.
3. Not all combinations of attributes are supported on all systems. For example, on a standard DOS PC, set to monochrome mode, REVERSE and UNDERLINE are mutually exclusive - only one of these attributes is actioned.

General Rules: for DISPLAY WINDOW statement Format 1 (DISPLAY WINDOW)

1. The DISPLAY WINDOW statement creates and makes current a terminal window. The terminal window is a rectangular region of your screen. Any ACCEPT or DISPLAY statements (apart from another DISPLAY WINDOW/LINE/BOX or a Format 1, 2 or 3 ACCEPT or Format 1 DISPLAY as described in your Language Reference) affect only the current window. Furthermore, line and column numbers for all ACCEPT and DISPLAY statements (apart from another DISPLAY WINDOW/LINE/BOX or a Format 1, 2 or 3 ACCEPT or Format 1 DISPLAY as described in your Language Reference) are computed from the upper left-hand corner of the current window. That is, the current window defines a virtual terminal screen which occupies some area of your physical screen.
2. The initial window is set to the entire screen.
3. The only way to change the current window is with another DISPLAY WINDOW statement or with the CLOSE WINDOW statement.
4. The LINE NUMBER phrase sets the top line of the window. Line number one refers to the top line of the screen. Line numbers are relative to the screen, and not to the current window.
5. If the LINE NUMBER phrase is not specified, is specified as zero, or is off the physical screen, the top line of the screen is used.
6. The COLUMN NUMBER phrase sets the left-most column of the window. Column number one refers to the left side of the screen. Column numbers are relative to the screen, and not to the current window.
7. If the COLUMN NUMBER phrase is not specified, is specified as zero, or is off the physical screen, column number one is used.
8. The SIZE phrase sets the number of columns the window contains. If this causes the window to extend past the right edge of the screen, the window's width extends off the screen.
9. If the SIZE phrase is not specified or is specified as zero, the window extends to the right edge of the screen.
10. The LINES phrase sets the number of rows the window contains. If this causes the window to extend past the bottom of the screen, the window extends off the screen.
11. If the LINES phrase is not specified or is specified as zero, the window extends to the bottom edge of the screen.
12. When the ERASE phrase is specified, the window is cleared immediately after it is created. Otherwise the window's contents are not changed. Clearing a window sets it to spaces.
13. The BOXED phrase causes a box to be drawn around the new window. The box is drawn outside the window. Any portions of the box that lie off the screen are not drawn.
14. The terminal's line drawing set is used to draw the box. If the terminal does not have a line drawing set, equivalent ASCII characters are used. If the POP-UP phrase is also specified, the box overlays any other boxes on the screen. If this phrase is not specified, the box drawn is attached to any other boxes it intersects. When a boxed non-popup window intersects a boxed popup window, if the popup window

is created first, when it is closed the points where the two window boxes intersected is not redrawn. That is, intersection characters remain even though there is no longer an intersection.

15. The ERASE phrase is implied by the BOXED phrase.
16. The REVERSED phrase exchanges the window's foreground and background colors. This affects every ACCEPT and DISPLAY statement in the new window.
17. The REVERSED phrase implies the ERASE phrase. This usually causes the entire window to be set to reverse video spaces when it is initially created.
18. The SHADOW phrase causes the window to appear to float over the screen giving a three-dimensional effect.
19. If the color value for either foreground or background is set to 0 in the COLOR field, then the corresponding color of the default system attribute is used.
20. The TITLE phrase causes the title to be printed in the window's border. This has its effect only if the BOXED phrase is also specified.
21. Titles can be placed in one of six positions in the border region: top left, top center, top right, bottom left, bottom center and bottom right. If TOP or BOTTOM is not specified, TOP is used. If LEFT, CENTERED or RIGHT is not specified, CENTERED is used.
22. The NO SCROLL phrase is treated as documentary only; the Windows preprocessor displays a message confirming this.
23. The NO WRAP phrase is treated as documentary only; the Windows preprocessor displays a message confirming this.
24. The POP-UP AREA phrase causes your COBOL system to save system information prior to creating the new window. This information can be used by the CLOSE WINDOW statement to subsequently remove the new window and restore the underlying windows. This gives a popup window.
25. The *save-area* data item is filled in with system information. This data item must not be subsequently modified in any way or results are undefined. It can be referenced in a CLOSE WINDOW statement to restore an earlier window to the screen and re-establish that window as the current window.

General Rules: for DISPLAY LINE statement Format 2 (DISPLAY LINE)

1. The DISPLAY LINE statement enables you to draw vertical and horizontal lines in a machine- and terminal-independent manner. The lines are drawn using the best mode available on the display device. Used together with the DISPLAY BOX statement, this provides the ability to draw forms on your screen. The DISPLAY LINE statement does not affect the positioning of full screen ACCEPT and DISPLAY statements.
2. Lines are drawn so that when they intersect other lines on the screen, the appropriate intersection character is used. This is done so that when the end of a line intersects another line, the appropriate corner or three-way intersection is used.
3. If the SIZE phrase is specified, the line drawn is horizontal. The value of *length* gives the size of the line in screen columns. If the LINES phrase is used instead, the line drawn is a vertical line and *height* describes the number of screen rows to use.
4. Lines never wrap around or cause scrolling. If the LINES or SIZE phrase would cause the line to leave the current window, the line is truncated at the edge of the window. If LINES or SIZE is zero, no line is drawn.
5. The value of *line-num* gives the starting row of the line. The value of *col-num* gives the starting column. Lines are always drawn to the right or downward as appropriate. *line-num* and *col-num* must specify a position that is contained in the current window.
6. If the LINE NUMBER or COLUMN NUMBER phrases specify a point outside the physical screen, that is, *line-num* = 0 or 24 (or your screen's maximum), or *col-num* = 0 or > 80, no line is drawn.
7. The TITLE phrase has effect only when drawing horizontal lines. When specified, *title-string* is printed in part of the line.
8. The title can be printed near the right side, near the left side or in the center of the line depending on the RIGHT, LEFT or CENTERED phrase specified. If none is specified, CENTERED is used.

9. The REVERSE phrase exchanges the foreground and background color of the line.

General Rules: for DISPLAY BOX statement Format 3 (DISPLAY BOX)

1. The DISPLAY BOX statement enables you to draw a box in a machine- and terminal-independent manner. The boxes are drawn using the best mode available on the display device. If the lines used in drawing a box intersect other lines already present on the screen, the appropriate intersection characters are used. The DISPLAY BOX statement does not affect the positioning of full screen ACCEPT and DISPLAY statements.
2. The location of the box is specified by providing the location of the upper-left corner. The size of the box is specified by providing a height and a width.
3. If the LINE NUMBER or COLUMN NUMBER phrases specify a point outside the physical screen no box is drawn.
4. The SIZE phrase specifies the width of the box. The LINES phrase specifies its height. If the SIZE phrase is not specified, or zero, or such that the box would extend beyond the physical screen or the edge of the window, the box extends to the right edge of the current window. If the LINES phrase is not specified, or zero, or such that the box would extend beyond the physical screen, the box extends to the bottom of the current window.
5. The REVERSE phrase operates in the same manner as it does for a DISPLAY WINDOW statement.
6. The TITLE phrase operates in the same manner as it does for the DISPLAY WINDOW statement.

Windowing Restrictions

- This feature is not guaranteed to be intermediate code compatible, so you might need to recompile your source code between product releases.
- When using the ACCEPT or DISPLAY statements with this windowing syntax, you must include the AT LINE NUMBER syntax (see your Language Reference) or items do not appear in the windows.
- You should not use cobprintf() with these DISPLAY statements.
- You should not use COPY REPLACING or REPLACE statements.
- The windowing syntax is supported only for fixed format COBOL source.
- The following reserved words have been introduced by the windowing syntax, so you should avoid specifying them as user-defined words:

  BOX BOXED CENTERED COLOR (if COLOR directive used) POP-UP SCROLLSHADOW WINDOW WRAP
- You should use only the ACCEPT and DISPLAY statements documented in your Language Reference with this windowing syntax.
- When using windowing syntax, the ANS85 Compiler directive is implied. You must not unset this directive either explicitly or implicitly.
- Alphanumeric literals must not be continued over the end of any line which includes a windowing statement.
- Some syntax errors, for example, spelling PROCEDURE DIVISION incorrectly, are flagged, but might result in spurious error messages for following source lines.
- Windowing syntax errors are serious errors, but are flagged in the form:

  ```
  xnnn-P*******
  ```
- The -P cob flag should not be used with windowing syntax . You should instead use "-C list".
- Column 73 must not be used within source programs which use windowing syntax, as this column is always treated as being set to a space character.
- The Compiler asks if you wish to continue after any error occurs. You can disable this function by using the NOERRQ directive. You should not, however, use the NOERRQ directive when compiling from within the Development Environment.

  If no error occurred, or if an error occurred but you replied "no" to the question "do you wish to continue", the Compiler returns a zero error return code.

- Each of the following statements must appear on a line by itself:

  DISPLAY WINDOW DISPLAY BOX DISPLAY LINE CLOSE WINDOW EXIT PROGRAM
- The windowing subsystem is initialized automatically upon encountering the first windowing statement.
- If an application switches between using windowing syntax and other types of Accept/Display syntax, it must close down the windowing system completely before starting to use other types of Accept/Display syntax; otherwise the ACCEPT and DISPLAY statements may not have the desired effects.

  You can create a subroutine to explicitly close the windowing system by compiling the following subprogram:

  ```
  $set preprocess "window1" autoclose
  procedure division
  para-1.
  exit program.
  ```

  You then call this subprogram before switching to another type of Accept/Display syntax. The AUTOCLOSE preprocessor directive causes the EXIT PROGRAM statement to close down the windowing system before exiting the subprogram. The windowing subsystem is reinitialized upon encountering another windowing statement. Each time the windowing subsystem initializes, the background screen and contents are redisplayed.
- When a window is active, or has been active in the run unit, use of the DISPLAY SPACES UPON CRT statement clears the window to spaces but leaves attributes unchanged.

Windowing Error Messages

The following errors might be encountered during preprocessing.

Unexpected numeric literal

Unexpected alphanumeric literal

Unsupported keyword or noise word

Unrecognized clause to DISPLAY WINDOW

Unrecognized clause to DISPLAY LINE

Unrecognized clause to DISPLAY BOX

Unrecognized clause to ACCEPT FROM SCREEN

This keyword has already been used

This keyword conflicts with another

This reserved word is used incorrectly

Wrongly formed or ordered clause with keyword

Error during preprocessing - no further details

Unknown COPY file specified

WINDOW1 preprocessor cannot handle free format

SCROLL/WRAP clause processed as comment

The edit/compile/animate loop returns to an incorrect line within your source program after returning an error.

Windowing Supplementary Information

When the first windowing statement in your program is encountered the screen is redisplayed. This is expected behavior and does not affect your program in any way.

Upgrading from RM/COBOL®

There are a number of settings in Visual COBOL that are designed specifically to ensure that your existing RM/COBOL source code can compile and run in Visual COBOL.

Refer to the *Compatibility with RM/COBOL* section for guidance and best practice on moving your applications to Visual COBOL. It covers:

- Supported RM/COBOL features, including detailed information on support for data types and subprograms.
- Syntactical differences between the two COBOL dialects, including workarounds or equivalent syntax where applicable.
- Details on how to configure your applications to continue using your RM/COBOL data files.

Compatibility with RM/COBOL

Visual COBOL provides compatibility with the RM/COBOL programming language:

This enables you to migrate programs from this environment. You can:

- Convert applications written in RM/COBOL to the Micro Focus COBOL language, and enhance them using the advanced language and development features offered by Visual COBOL.
- Retain the use of the selected COBOL on some machine environments while moving to Visual COBOL on others. You might want to maintain a common set of source programs which are suitable for all environments.

Note: Any error messages and numbers that are returned when you compile your program in Visual COBOL or when you execute the resulting code are different in the two environments. This should present no problems, but is something of which you should be aware.

Converting RM/COBOL Applications

By default, this COBOL system already supports much of the RM/COBOL syntax and behavior. Additional RM/COBOL-specific syntax that has been added for compatibility is documented in the section *RM/COBOL Syntax Support* in your *Language Reference*.

You can also enable additional RM/COBOL behavior using certain Compiler directives. Using these directives when you submit your RM/COBOL source programs to this COBOL system ensures that most of the programs are accepted the first time they are submitted. There are still certain compatibility issues between the two COBOL systems, which are detailed, with any possible workarounds, in the *RM/COBOL Conversion Issues* section.

Compiler Directives for RM/COBOL Compatibility

You can set a number of Compiler directives in your RM/COBOL source programs that enable a program to emulate RM/COBOL behavior.

The main directive that sets the majority of RM/COBOL behavior is DIALECT(RM).

Setting the DIALECT(RM) directive automatically sets additional Compiler directives, such as RM, NOTRUNC, OLDINDEX, NOOPTIONAL-FILE, RETRYLOCK, ALIGN"2" and SEQUENTIAL"LINE". See the topic *RM Dialect Settings* for full details of the directives set.

The system will also behave as if you had specified the following syntax:

```
sign trailing separate
```

for signed numeric data items, and:

```
lock mode is automatic
```

Previously, compatibility was achieved by compiling with the RM Compiler directive. The newer DIALECT(RM) directive sets and extends the compatibility given by RM, and should be used for all new migrations from RM/COBOL, unless you normally set the ANSI switch when you submit your RM/COBOL source programs to the RM/COBOL system. If you do, set the RM"ANSI" directive when you compile your programs.

We also recommend that you set the NOMF directive when you submit your RM/COBOL source programs to this COBOL system. This ensures that only those words which are treated as reserved words under the ANSI '74 COBOL standard are regarded as reserved words by this COBOL system.

Setting the NORM directive disables the syntax enabled when the RM directive was set, and automatically resets the additional Compiler directives to NOSPZERO, TRUNC"ANSI", NOOLDINDEX, OPTIONAL-FILE, NORETRYLOCK, ALIGN"8" and SEQUENTIAL"RECORD". Additionally, the system behaves as if you had specified the syntax:

```
sign trailing included
```

for signed numeric data items, and:

```
lock mode is exclusive
```

for each file in the program which has no explicit locking syntax declared.

The final states of the additional directives set when you use the NORM directive are not necessarily the same as their initial default states.

Converting RM/COBOL Data Types

Types of Data

This COBOL system always allocates the same number of bytes to a data item as in the original program, but might redefine its contents when it is converted. Therefore, you must be aware that if these items are redefined, and the program logic expects to find a certain binary value in the redefinition, you might not receive the behavior you are expecting at run time. At run time, this COBOL system might treat the ON SIZE ERROR clause differently from the RM/COBOL system. See the chapter *RM/COBOL Conversion Issues* for further details.

The following sections define how this COBOL system treats COMPUTATIONAL, COMPUTATIONAL-1 and COMPUTATIONAL-6 types of data.

COMPUTATIONAL (COMP) Data Types

This COBOL system treats any COMP data items in your RM/COBOL source program as the standard Micro Focus COBOL DISPLAY format. The difference in the internal representation of such data in the two systems is that this COBOL system always sets the most significant four bits of each byte to the value 3, while the RM/COBOL system always sets such bits to the value 0.

For example, under the RM/COBOL system:

```
PIC 999 COMP VALUE 123
```

is held in three bytes as hexadecimal value 01 02 03

while under this COBOL system:

```
PIC 999 VALUE 123
```

is held in three bytes as hexadecimal value 31 32 33.

COMPUTATIONAL-1 (COMP-1) Data Types

If you compile your RM/COBOL source programs with the *DIALECT"RM"* or *COMP1"BINARY"* Compiler directives, each data item declared as USAGE COMP-1, regardless of its picture-string, is interpreted as a 2-byte signed binary data item in this COBOL system, which is a standard Micro Focus COBOL picture-

string of s9(4) COMP. This data item is capable of holding a hexadecimal value in the range -32768 to +32767.

Without an appropriate Compiler directive in place, this COBOL system interprets COMP-1 data items as single-precision floating point data items.

See your *Language Reference* for details on the standard Micro Focus COBOL language.

PICTURE Character-strings

The maximum length of a PICTURE character-string in a record description is 20 characters. However, you can overcome this limitation by splitting any PICTURE character-string which exceeds this limit into two, and defining a FILLER item with a PICTURE character-string which corresponds to the size of the second half of the original string.

COMPUTATIONAL-6 (COMP-6) Data Types

This COBOL system treats any COMP-6 data items in your RM/COBOL source program as the standard Micro Focus COBOL COMP format. If, as a result of this, less data space is allocated to each item than would be under the RM/COBOL system, this COBOL system pads the space with a byte containing a zero, as shown in the following table:

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause

[illegible]

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	[illegible]

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	

RM COMP-6 PICTURE clause	

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	

RM COMP-6 PICTURE clause	

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	[illegible]

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	
	Number of bytes containing leading zeros required to precede COMP PICTURE claus [illegible]

RM COMP-6 PICTURE clause

RM COMP-6 PICTURE clause	[illegible]

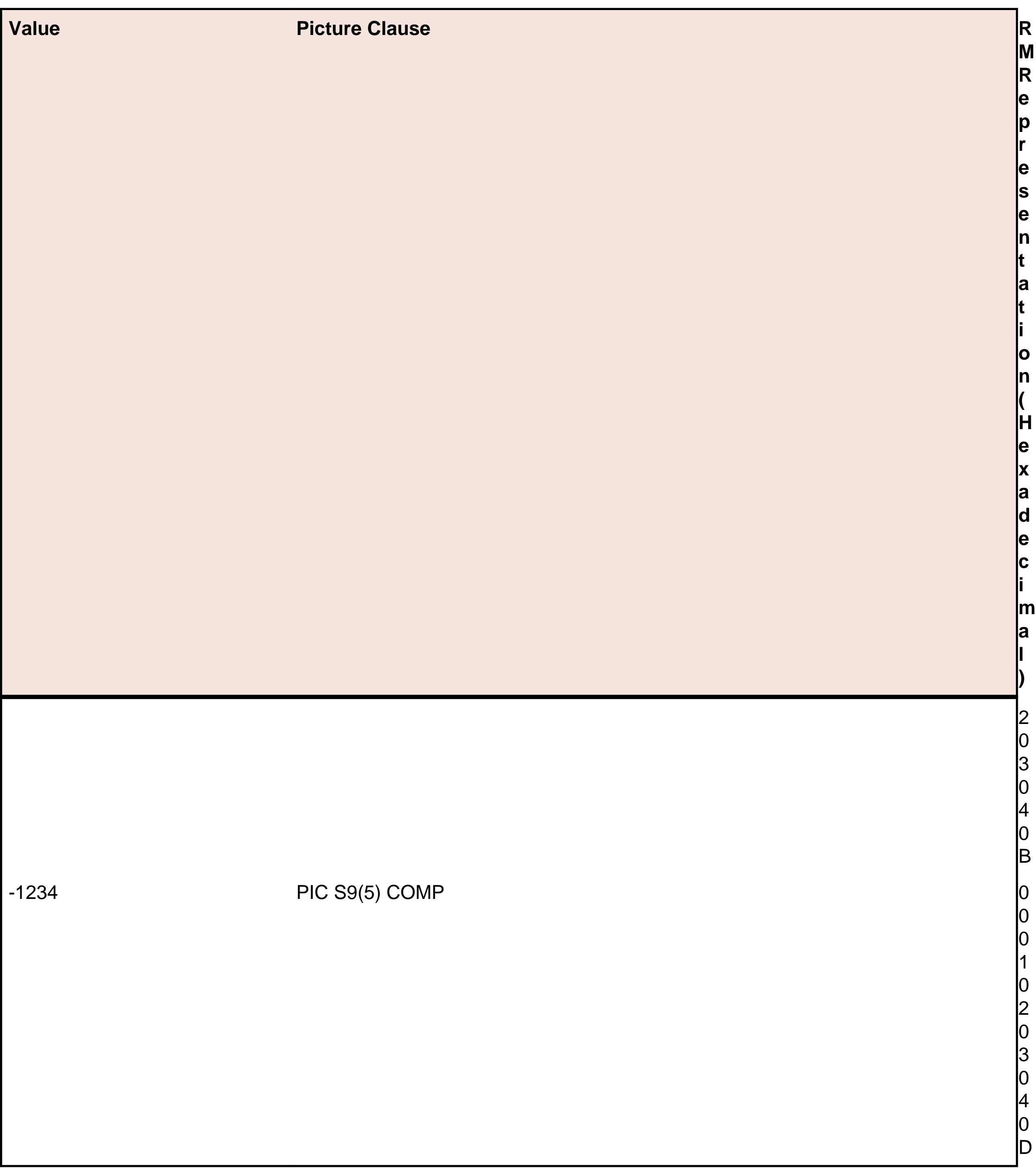
The padding byte precedes the converted field, and is therefore included in any redefinition of group fields which contain the converted field. However, the padding byte would not be included if you redefined only the converted field. If you wish to ensure that the padding byte is included in the redefinition of the converted field, you must add a field at a higher level immediately before the converted field, and redefine this new field instead.

COMP/COMPUTATIONAL Data

The RM/COBOL system represents COMP (or COMPUTATIONAL) data in packed decimal format with one character per byte stored in each least significant four bits. The most significant half-byte always contains zero. If the picture string specifies a signed representation, an additional byte is added to the least significant end of the string: a negative value is represented by the hexadecimal value x"D", and a positive value is represented by the hexadecimal value x"B".

Consider the following examples:

Value	Picture Clause	RM Representation (Hexadecimal)
1234	PIC 9(5) COMP	0001020304
1234	PIC S9(5) COMP	00010

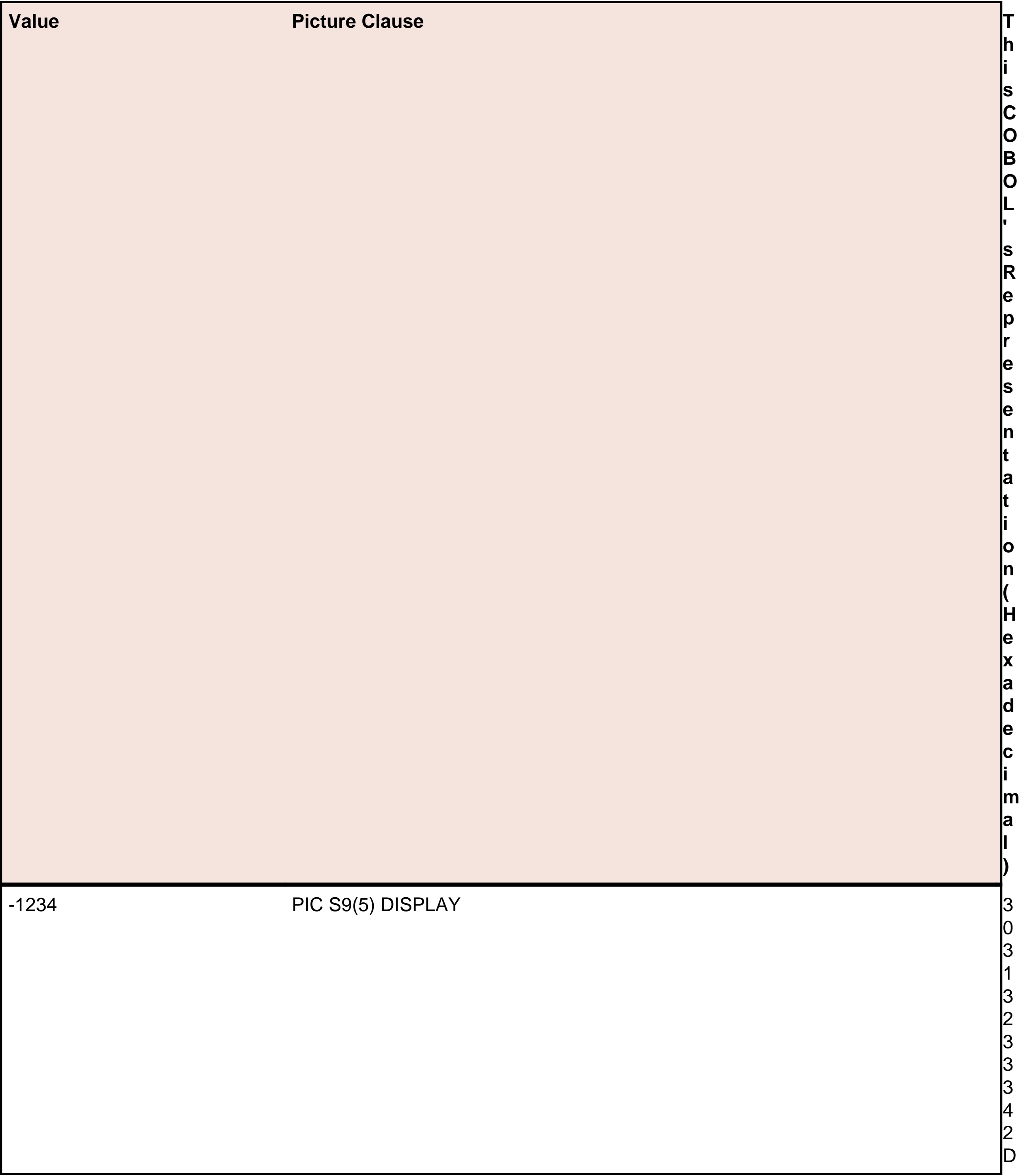


Value	Picture Clause	RM Representation (Hexadecimal)
-1234	PIC S9(5) COMP	203040B00010203040D

You should converts COMP data fields into DISPLAY format, with sign trailing separate. This is compatible with this COBOL system's treatment of RM/COBOL COMP fields in the source program when the RM directive is set. If the data item is signed, the sign byte has the most significant half-byte set to hexadecimal value 2.

After conversion, the examples above are represented as follows:

Value	Picture Clause	This COBOL's Representation (Hexadecimal)
1234	PIC 9(5) DISPLAY	3031323334
1234	PIC S9(5) DISPLAY	30313233342B



Value	Picture Clause	This COBOL's Representation (Hexadecimal)
-1234	PIC S9(5) DISPLAY	30313233342D

COMP-3/COMPUTATIONAL-3 Data

The RM/COBOL system represents COMP-3 data in packed decimal format with the least significant half-byte holding the sign.

This sign half-byte contains the following values:

Field	RM Sign half-byte value (Hexadecimal)
Unsigned	F

Field	RM Sign half-byte value (Hexadecimal)
Signed, Positive	B or F
Signed, Negative	D

Consider the following examples:

Value	Picture Clause	RM Representation (Hexadecimal)
1234	PIC 9(5) COMP-3	00234F
1234	PIC S9(5) COMP-3	00234F
-1234	PIC S9(5) COMP-3	00234D

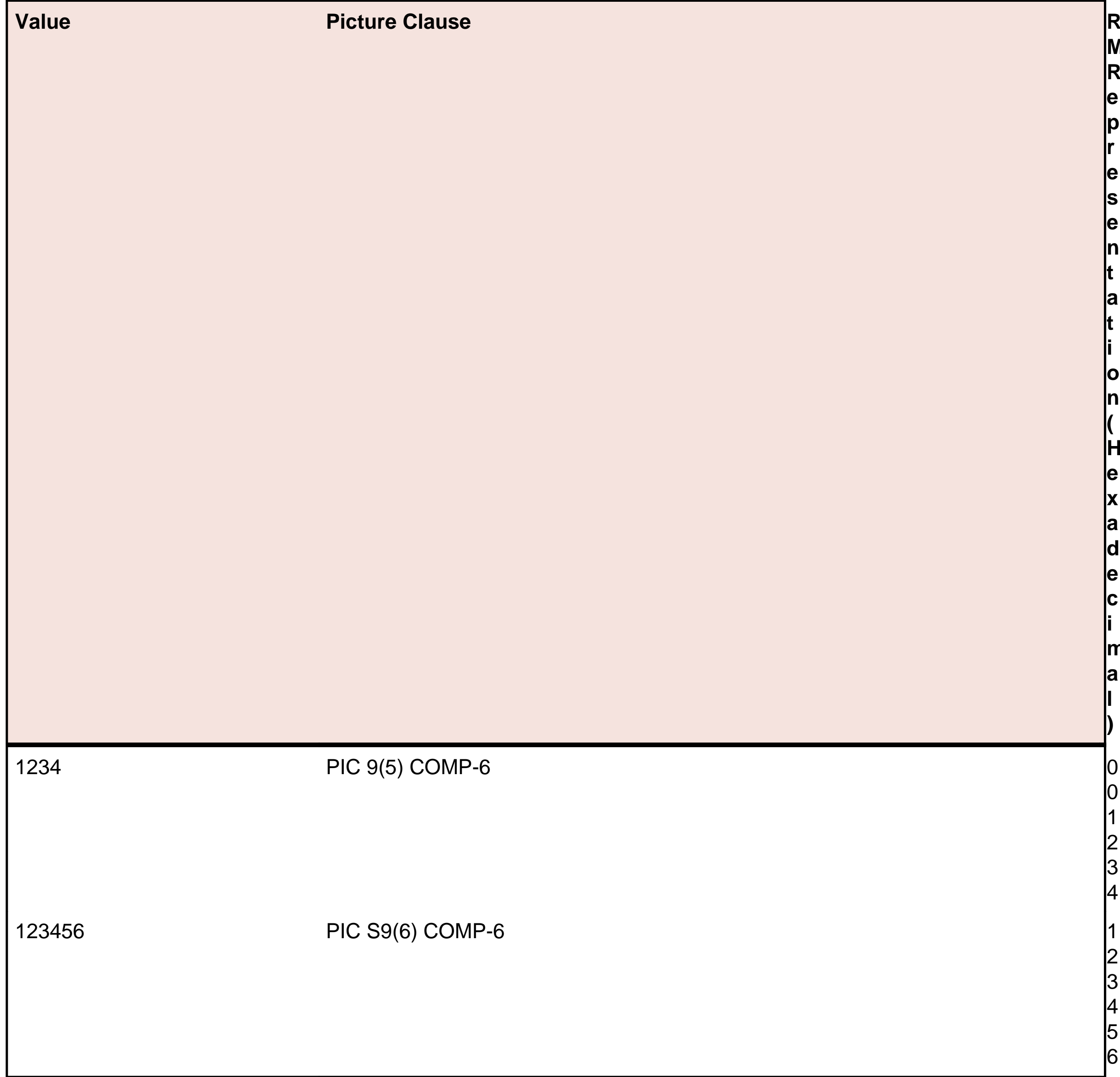
The only requirement for conversion is that the sign half-byte has to be changed for signed positive fields to hexadecimal value C. The examples above are represented as follows:

Value	Picture Clause	This COBOL's Representation (Hexadecimal)
1234	PIC 9(5) COMP-3	00234F
1234	PIC S9(5) COMP-3	00234C
-1234	PIC S9(5) COMP-3	00234D

COMP-6/COMPUTATIONAL-6 Data

The RM/COBOL system holds COMP-6 data in a similar format to COMP-3 data, except there is no sign half-byte. If a sign is indicated in the picture clause it is ignored and has no effect. The value held is always positive.

Consider the following examples:



Value	Picture Clause	RM Representation (Hexadecimal)
1234	PIC 9(5) COMP-6	001234
123456	PIC S9(6) COMP-6	123456

In order to maintain the size and capacity of the data items, this COBOL system treats COMP-6 data items as this COBOL's COMP format fields, and pads the field with binary zeros where necessary. The examples above are represented as follows:

Value	Picture Clause	This COBOL's Representation (Hexadecimal)
1234	PIC 9(5) COMP-6	0004D2
123456	PIC S9(6) COMP-6	01E240

The 9(5) COMP field is extended by one byte containing binary zero in order to maintain the size of the original item. See the section *COMPUTATIONAL-6 (COMP-6) Data Types* in the chapter *Compatibility with RM/COBOL* for details.

DISPLAY Data

You should be aware of the following differences between the representation of numeric DISPLAY format data items, with sign INCLUDED, under the RM/COBOL system and this COBOL system:

- For positive values, this COBOL system does not encode a sign on the data, whereas the RM/COBOL system increments the value of the most significant half-byte by one to denote a positive value.
- For negative values, this COBOL system increments the value of the most significant half-byte by four, whereas the RM/COBOL system increments the value of the byte by twenty-five (that is, hexadecimal 19).

Use the *SIGN* directive with the `EBCDIC`, `IBM` or `NCR` parameters to maintain the same RM/COBOL representations in this COBOL system.

When the *DIALECT"RM"* directive is set, a DISPLAY format data item with no sign clause associated is treated by this system as though you had specified the SIGN TRAILING IS SEPARATE clause. However, when the *RM"ANSI"* directive is set, the same data item is treated as though you had specified the SIGN TRAILING IS INCLUDED clause. This is the default state for such data items in this COBOL system.

Consider the following examples:

Value	Picture Clause	RM/COBOL's Representation (Hexadecimal)	
		Leading	Trailing
123	PIC 9(3) DISPLAY	31 32 33	31 32 33
123	PIC S9(3) DISPLAY	41 32 33	31 32 43
-123	PIC S9(3) DISPLAY	4A 32 33	31 32 4C

These examples are represented as follows:

Value	Picture Clause	This COBOL's Representation (Hexadecimal)	
		Leading	Trailing
123	PIC 9(3) DISPLAY	31 32 33	31 32 33
123	PIC S9(3) DISPLAY	31 32 33	31 32 33
-123	PIC S9(3) DISPLAY	71 32 33	31 32 73

RM/COBOL Conversion Issues

The syntax of most RM/COBOL source programs when submitted to run on this COBOL system will be accepted and run successfully. However, sometimes this COBOL system might reject some of the syntax in your original RM/COBOL source program, or might cause your program to behave unexpectedly at run-time.

This section contains the known problems which you may encounter. Hints are also given on how you can either rectify the cause of such errors, or emulate the RM/COBOL type of behavior in this COBOL system.

Producing Executable Code

The following section covers the known issues when submitting RM/COBOL source programs to this COBOL system. Where possible, work-arounds and resolutions are also provided.

Perform Statements

PERFORM statements are not treated in the same way by both COBOL systems. This COBOL system uses a stack-based perform handling system, while the RM/COBOL system associates a return address with a specific procedure name.

As a result, under the RM/COBOL system, all end-points to PERFORM statements are always active until they are used. However, under this system, only the end-point of the last PERFORM statement is active at any one time.

You must set the PERFORM-TYPE directive with the RM parameter if this COBOL system is to emulate the behavior of RM/COBOL PERFORM statements.

ACCEPT FROM CENTURY-DATE and FROM CENTURY-DAY
In Visual COBOL, to use the FROM CENTURY-DATE and FROM CENTURY-DAY phrases with the ACCEPT statement, set the RM Compiler directive.

Alternatively, use the following equivalent phrases with the ACCEPT statement:

- FROM DATE YYYYMMDD, which is the equivalent of FROM CENTURY-DATE.
- FROM DAY YYYYDDD, which is the equivalent of FROM CENTURY-DAY.

```
procedure division.
ACCEPT data-name-1 FROM DATE YYYYMMDD.
ACCEPT data-name-2 FROM DAY YYYYDDD.
```

Nested COPY statements with REPLACING phrase
In Visual COBOL, you cannot specify text replacement as part of a nested COPY statement when text replacement is already active as part of a COPY statement.

If you attempt to use COPY REPLACING in a file copied with a COPY REPLACING statement, an error code *COBCH0062 COPY replacement not supported* is displayed on compilation.

Duplicate Paragraph-names

In Visual COBOL, if you have duplicate paragraph-names, in different sections, and then call a paragraph-name from outside its section, an error is produced unless you have explicitly referenced the paragraph-name and its section. In RM/COBOL, by just calling the paragraph-name, it assumes you are calling the next declaration of the paragraph-name found.

To ensure that references to duplicate paragraph-names are correctly resolved, you must qualify a reference to a duplicate paragraph-name by adding the section-name in which it is declared.

Example

If your source code contains the following:

```
      ....
      perform para-2.
      ....
 sect-1 section.
      para-1.
      ....
      para-2.
      ....
 sect-2 section.
      para-2.
      ....
```

RM/COBOL will resolve the reference to para-2 in the PERFORM statement by using the declaration of para-2 in the sect-1 SECTION. In Visual COBOL, however, you must qualify the reference to the duplicate paragraph-name in your source code by using the PERFORM para-2 OF sect-1 statement.

Empty Groups
In Visual COBOL, if you have any empty groups specified in your source code, you must set the DIALECT"RM" Compiler directive.

Figurative Constants and the USING Phrase
In Visual COBOL, to use figurative constants in the USING phrase of a CALL statement or as values of level 78 constants, set the DIALECT"RM" Compiler directive.

Alternatively, the figurative constant can be replaced by the equivalent non-numeric literal, such as " " for SPACE or "0" for ZERO.

File Not Found Errors
Visual COBOL and RM/COBOL differ in the environment variables that they use to locate program and data files.

If your source code produces a file not found error, ensure the correct paths are set in the correct environment variables. In Visual COBOL, set `COBPATH` to locate program files and `COBDATA` to locate data files. The `RUNPATH` environment variable used in RM/COBOL, is not used in Visual COBOL.

Indexed File Error on Open
Visual COBOL and RM/COBOL differ in how they handle record length fields and some data fields when you open an RM/COBOL indexed file.

In Visual COBOL, when you try to open an RM/COBOL indexed file, you may receive either a run-time error `COBRT161 Illegal intermediate code` or a file status code 39 `A conflict has been detected between the fixed file attributes and the attributes specified for that file in the program`.

You must ensure that you read in the same size records that were created in RM/COBOL.

In Visual Studio, hover over the level 01 item of the file description to display the length of the record.

If the length of the file description does not match that which was processed in RM/COBOL, check the following:

- In RM/COBOL, you can set the `RECORD CONTAINS nn CHARACTERS` clause to be a different length than the actual length specified in the record description. If this clause is greater than the actual description, you must pad the record description with filler bytes to match the `RECORD CONTAINS` clause.
- If you have signed numeric display data in your file, Visual COBOL will treat the sign as a separate byte if you are using the RM directive without "ANSI" specified. If these fields are stored as sign internal, you must use RM"ANSI" or do not use the RM directive at all.

Linkage Section in Main Program

In RM/COBOL, if the main program has a Linkage Section, it is initialized by the parameter passed on the command line. In Visual COBOL, you must use the *command_line_linkage* tunable to pass parameters from the command line to the Linkage Section.

Library Routines
To use the RM/COBOL standard library routines that are available in Visual COBOL, you must use the 1024 calling convention.

C$Century
C$ConvertAnsiToOem
C$ConvertOemToAnsi
C$DARG (only partially supported)
C$Delay
C$GetEnv
C$GetNativeCharset
C$LogicalAnd
C$LogicalComplement
C$LogicalOr
C$LogicalShiftLeft
C$LogicalShiftRight
C$LogicalXor
C$NARG
C$SetEnv

C$RERR
DELETE
RENAME

For more information on each library routine, see *RM/COBOL Library Routines* in the product Help.

Nested OCCURS DEPENDING Clauses

In Visual COBOL, if you are using nested OCCURS DEPENDING clauses, you must set the `ODOSLIDE` Compiler directive.

Numbering of Segments

In Visual COBOL, you can only specify segment numbers in the range 0 to 99 inclusive, which conforms to segment number limit specified in the ANS X3.23-1985 COBOL standard. In RM/COBOL, you can specify segment numbers greater than 99.

If your source code has segment numbers greater than 99, recode the program. Make sure that any new segment numbers you allocate do not clash with an already existing segment number. Segment numbers between 0 and 49 inclusive are used by Visual COBOL to indicate fixed portions of your object program, while segment numbers 50 to 99 inclusive indicate independent segments.

For details on the use of segmentation and segment numbers in your source programs, see *COBOL Language Reference* in the product Help.

Program Identification and Data Names

In Visual COBOL, you cannot use the same name for the Program-ID and a data item in the program; each name should be unique. RM/COBOL permits the name of the Program-ID paragraph and a data item to share the same name.

REMARKS Paragraph

In Visual COBOL, if your program uses the REMARKS paragraph in the Identification Division, you must set the DIALECT"RM" Compiler directive.

Alternatively, mark the paragraph as comment lines.

Reserved Words

In Visual COBOL, setting certain Compiler directives (such as RM and ANS85) activates certain reserved words that you cannot use as names for your data items.

If you attempt to use a reserved word, you receive a `COBCH0666 ("Reserved word used as data name or unknown data description qualifier")` COBOL syntax error.

To continue to use the reserved word as a data name, you can:

- Use the REMOVE Compiler directive, to unreserve that particular keyword.
- Set the MFLEVEL Compiler directive to an appropriate level, to unreserve all keywords above that level.

  See *Compiler Directives* in the product Help.

Example

Your RM/COBOL source program may contain the following lines of code:

```
    ....
    03 sort   pic 99.
    ....
    move 1 to sort
```

If you submit this to Visual COBOL, you will receive an error because the SORT verb is reserved. However, if you specify the REMOVE"SORT" Compiler directive when you submit this source program, you will not receive the error.

SAME AS Clause Not Available When Defining Data Structures

Visual COBOL and RM/COBOL differ in the way that they allow you to reuse existing data structures.

In Visual COBOL, use the TYPEDEF clause to define your base data structure, and then use the USAGE clause to create data structures of the same type.

```
data division.
working-storage section.
01 atype is typedef.
   03 var1 pic x(10) value "brown".
   03 var2 pic x(10) value "blue".
   03 var3 pic x(10) value SPACE.
01 a1 usage atype.
procedure division.
display var2 of a1.
```

The result of the display statement is `blue`.

The SAME AS clause used in RM/COBOL is not supported in Visual COBOL.

Source Code in Columns 73 to 80

Visual COBOL ignores any of the code in your source programs which lies within columns 73-80 inclusive.

Code in these columns could be the result of expanding TAB characters in your source program, instead of standard TAB stops. If your source program contains TAB stops, convert them to spaces.

Undeclared Data Items in Clauses

In Visual COBOL, you receive a `COBCH0250 STATUS field data-name missing or illegal` error if a data item used in the File Status clause is not declared in the Working-Storage section. In RM/COBOL, you do not have to declare the data item in Working Storage.

User-names Longer than 127 Bytes are Truncated

In RM/COBOL, you can specify user-names (data-names, procedure-names, program-names, etc) up to 240 characters long. In this COBOL system, user-names longer than 127 bytes in length are truncated and a warning message is produced.

Solution:

Results may be affected if the truncated user-name is used with XML Extensions, to export or import XML documents; therefore, we recommend that you keep user-names to 127 bytes or less.

Running the Code

Once you have successfully submitted your RM/COBOL source program to this COBOL system and produced executable code, you might encounter difficulties when you try to run this code under this system. Alternatively, the code might run but you might find that its behavior under this COBOL system is not exactly the same as under the RM/COBOL system. The following sections detail known areas of difficulty you might encounter, and offer hints on how you can avoid them.

Table Bound Checking

If you try to run a program under this COBOL system which contains a subscript value greater than the size of the table to which it refers, the run-time system will produce an error indicating this. Under the RM/COBOL system, however, no such table bound checking is done.

Therefore, if you wish to disable table bound checking in this COBOL system, you must use the NOBOUND directive.

If you use the NOBOUND directive when running intermediate code, you will be able to access data beyond a table's bounds by using a subscript value greater than the table size. Use of the NOBOUND directive when you are producing intermediate code will also disable bound checking when running

generated code. However, if you wish to access data beyond a table's bounds when running generated code, you must also use the directive NOBOUNDOPT.

Note: When you use the NOBOUNDOPT directive, performance will be impaired.

ACCEPT Fields at the Edge of the Screen

If your program contains an ACCEPT statement for a numeric data item at a position on the screen where the definition of the numeric data item would cause the ACCEPT field to go beyond the right-hand edge of the screen, both COBOL systems will truncate the input value. In RM/COBOL, the input value will be aligned into the ACCEPT field as an alphanumeric field, whereas in Visual COBOL, the input value is aligned as a numeric field.

Change the definition of the relevant PICTURE clause from numeric to alphanumeric. Alternatively, change the PICTURE clause so that the field does not go beyond the edge of the screen.

Example

If your program contains the following statement:

```
ACCEPT data-item AT COLUMN NUMBER 75.
```

where *data-item* is a numeric data item defined as PIC 9(10), a value of `123456` entered into the ACCEPT field will be held under Visual COBOL as `"0000123456"`. However, in RM/COBOL, the value in the ACCEPT field would be held as `"1234560000"`. To allow Visual COBOL to emulate the RM/COBOL behavior, alter the definition of the data item in your program to PIC X(10) or PIC 9(6).

COMPUTATIONAL-1 Data Items with a PICTURE other than S9(4)

In Visual COBOL, a COMPUTATIONAL-1 data item is assumed to have a picture-string of S9(4). In RM/COBOL, when a COMP-1 item is used as the source of a MOVE statement to an alphanumeric item, the picture clause is preserved.

To produce the result you require, you must alter the definition of the target of the MOVE statement.

Example

The following source code causes TEST-RECORD to hold`"99  "` in RM/COBOL, but`"0099"` in Visual COBOL.

```
 01 test-record                pic x(4).
 01 comp-1-item                pic 99 comp-1.
procedure division.
   ...

    move 99 to comp-1-item.
    move comp-1-item to test-record.
```

To overcome this problem, alter the definition of TEST-RECORD as shown below:

```
 01 test-record.
     03 test-numeric-field     pic 99.
     03 filler      pic xx.
 01 comp-1-item                pic 99 comp-1.
procedure division.
     ...
    move 99 to comp-1-item.
    move comp-1-item to test-numeric-field.
```

This avoids moving the COMPUTATIONAL-1 data item directly to an alphanumeric field.

Display of Input Data in Concealed ACCEPT Fields

If you have specified OFF and ECHO clauses for the same ACCEPT statement in your program, RM/COBOL will conceal any data entered during input for that statement but on completion of input will display

the data. Visual COBOL will not display the data for this ACCEPT statement once input has been completed.

If you want to display the data input for an ACCEPT statement with the OFF and ECHO clauses specified, you must add a DISPLAY statement after the ACCEPT statement.

Embedded Control Sequences in DISPLAY Statements

In Visual COBOL, you cannot embedded control sequences within data items that you want to be displayed.

Such characters are ignored at run time as they are hardware dependent.

Remove the control sequences from your source program and replace with the equivalent Micro Focus COBOL syntax; for example, use the syntax WITH UNDERLINE to replace `<left-arrow>]4m`.

End of File Notification

The first time you unsuccessfully try to READ a sequential file in either COBOL system because you have reached the end of the file, status key 1 in the FILE STATUS is set to 1 and status key 2 is set to 0. This indicates that there is no next logical record. If you try to READ the same file again, without it either having been previously closed and reopened, or it having been successfully started, Visual COBOL continues to indicate that there is no next logical record. However, if you try to READ the same file again under RM/COBOL, status key 1 is set to 9 and status key 2 is set to 6.

A solution to the different file statuses returned for the circumstances given above will depend on the way in which your source program is coded. We suggest that you include tests for the values 1 and 0 in status key 1 and 2 of the file status, respectively, at the same time as you test for the values 9 and 6 in these status keys.

Field Wrap-Around

If, when using binary data items (that is, RM/COBOL COMPUTATIONAL-1 format items) an arithmetic operation gives a value which exceeds the capacity of the data item, and there is no ON SIZE ERROR clause, Visual COBOL wraps-around the value of the item. However, in RM/COBOL, the data item is set to the limit of its capacity.

You should specify an ON SIZE ERROR clause to highlight such problems.

Example

In RM/COBOL, the following lines of code result in the value +32767 being stored in the data item, CALC-ITEM. However, Visual COBOL sets CALC-ITEM to -32768:

```
01 calc-item         pic s9(4)  comp-1.

 procedure division.
      ....
     move 32767 to calc-item.
     add 1 to calc-item.
```

File and Record Locking

Certain versions of RM/COBOL contain some software errors in the way in which locks for files and records are handled. These errors do not occur in Visual COBOL.

The errors fixed when upgrading to Visual COBOL are:

- Indexed files do not detect or acquire locks if they are opened for output. This is regardless of whether you specify the WITH LOCK phrase
- Relative and sequential files cannot be locked exclusively
- Files which are opened for input can detect record locks, although the RM/COBOL documentation states that they cannot. When the RM directive is set in Visual COBOL, record locks can still be detected by files opened for input

- The first record in a sequential file opened for input-output is locked whenever any other record in that file is locked

Initialization of Working Storage

Visual COBOL initializes all working storage items without VALUE clauses to SPACES .The RM/COBOL system initializes all working storage items to SPACES, unless you have placed numeric data items between data items with VALUE clauses.

If this feature causes you any problems, because your program relies on the initial value given to the system, add a VALUE clause with the appropriate value to your source program and resubmit it.

Example

The RM/COBOL system initializes the following group item to SPACES:

```
01 group-item.
     03 item-1     pic x.
     03 item-2     pic 99.
     03 item-3     pic x.
```

However, if item-1 and item-3 have value clauses associated with them, the RM/COBOL system initializes the second byte of item-2 to hexadecimal value 0 when item-2 is defined as USAGE COMP (signed or unsigned) or USAGE DISPLAY (unsigned only).

Numeric Fields Containing Illegal Characters When Using a DEPENDING ON Phrase of an OCCURS Clause

In Visual COBOL, if you fail to initialize a numeric data item that is used in a DEPENDING ON phrase of an OCCURS clause appropriately, a `COBRT163 Illegal character in numeric field` error is displayed at run time, because the data item is initialized to SPACES if no value is specified. In RM/COBOL, the data item is initialized to ZERO, and therefore, the error does not occur.

ON SIZE ERROR Phrase

In Visual COBOL, the ON SIZE ERROR condition exists when the value resulting from an arithmetic operation exceeds the capacity of the specified picture-string. In RM/COBOL, the ON SIZE ERROR condition exists when the value resulting from an arithmetic operation exceeds the capacity for the associated data item.

Ensure that the capacity of any data items in your source programs is specified by a picture-string; for example, COMPUTATIONAL-1 data items.

Open EXTEND of Nonexistent File

In Visual COBOL, because setting the RM Compiler directive sets the NOOPTIONAL-FILE Compiler directive, if you try to open a non-existent file for I-O or EXTEND the run-time system will give an error message. For I-O, RM/COBOL does the same. However, for EXTEND, RM/COBOL creates the file and opens it as if you had specified OUTPUT.

The following options are available:

- Add the keyword OPTIONAL to the SELECT statement. This makes Visual COBOL create the file and open it for OUTPUT
- Create the empty file before running your program
- Specify the OPTIONAL-FILE Compiler directive. This makes Visual COBOL create the file and open it for OUTPUT. However, the behavior with files opened for I-O will now differ from RM/COBOL

Printer Output is Written to Disk

By default, Visual COBOL writes all output intended for a printer to disk.

To send output to a physical printer, you must map the filename using the dd_LPT1 environment variable or, if your system supports the lp printer spooler, you should use:

```
dd_LPT1=">lp";export dd_LPT1
```

Redefinition of COMP or COMP6 Data Items

Visual COBOL fully supports the size and capacity of RM/COBOL type COMPUTATIONAL and COMPUTATIONAL-6 data items when the RM Compiler directive is set. However, the internal representation of these data items in Visual COBOL and in RM/COBOL is not the same. See the chapter *Compatibility with RM/COBOL* in the product Help.

If this causes you problems, redefine these data items to take advantage of their internal format. MOVE the data items concerned to other data items which are not defined as COMPUTATIONAL or COMPUTATIONAL-6. Moving these data items converts the data automatically, overcoming any problems you might have.

Example

The following source code is coded to take advantage of the internal representation of COMPUTATIONAL-6 data items in RM/COBOL, and to analyze a date field:

```
 01 birthdate-1          pic 9(6) comp-6.
 01 birthdate-2 redefines birthdate-1.
     03 month-2         pic 99 comp-6.
     03 day-2           pic 99 comp-6.
     03 year-2          pic 99 comp-6.
     ....

procedure division.
start-up section.
para-1.
    ....

    move 082462 to birthdate-1.
    ....

    if year-2 = 62
        display "records not available for 1962."
```

Amend your source program to use the DISPLAY format instead of redefining COMPUTATIONAL-6 data items, before submitting it to Visual COBOL:

```
 01 birthdate-1          pic 9(6) comp-6.
 01 birthdate-2 redefines birthdate-1.
     03 month-2         pic 99 comp-6.
     03 day-2           pic 99 comp-6.
     03 year-2          pic 99 comp-6.

 01 birthdate-1a         pic 9(6).
 01 birthdate-2a redefines birthdate-1a.
     03 month-2a        pic 99.
     03 day-2a          pic 99.
     03 year-2a         pic 99.
     ....

procedure division.
start-up section.
para-1.
    ....

    move 082462 to birthdate-1.
    move birthdate-1 to birthdate-1a.
    ....
```

```
     if year-2a = 62
         display "records not available for 1962."
```

Screen Column Number Specification

Visual COBOL permits you to specify screen column numbers up to and including 999, but RM/COBOL permits you to specify column numbers greater than 999. If you try to run an RM/COBOL source program containing a column number greater than 999 in Visual COBOL, the column number is truncated so that only the last three digits are used. If truncation of the column number occurs for an item to be displayed on the screen, the position of that item on the screen in Visual COBOL will differ from its position in RM/COBOL.

Trailing Blanks in Line Sequential Files

Visual COBOL always removes trailing blanks from line sequential records before writing the record. RM/COBOL removes trailing blanks from such records only if the FD entry contains 01 level records of different sizes. This will not cause you any problems when you run your converted RM/COBOL programs in Visual COBOL. However, you may receive errors at run time if any REWRITE operations on line sequential files change the length of the records.

Change the file organization to sequential, or move an alternative padding character (for example, LOW-VALUES) to the end of the record before it is written. This ensures that full-length records are written.

You also need to ensure that the T run-time switch is not set, as this might also change the size of the record. See *Run-time Switches* in the product Help.

Undefined Results of MOVE and Arithmetic Operations
Visual COBOL and RM/COBOL differ in the results of MOVE statements, arithmetic operations, and comparisons that involve numeric and alphanumeric data items.

You can overcome most of these incompatibilities by redefining the data items involved, or by recoding the comparisons. If you submit a program in Visual COBOL containing an alphanumeric to numeric data item MOVE statement, a warning message will be displayed indicating this.

Example

If you submit a source program containing the following data items and procedural statements, the specified test will fail at run time:

```
 01 numeric-field        pic 9(5).
procedure division.
    move "abc" to numeric-field.
    if numeric-field = "00abc"
        ....
```

When the RM Compiler directive is set, Visual COBOL partially emulates the behavior of RM/COBOL for alphanumeric to numeric MOVEs by treating the numeric item as an alphanumeric item which is right justified. However, the above example will still fail because RM/COBOL treats the literal ABC as numeric, and places 00ABC in the numeric item. To make the statement run successfully in Visual COBOL, amend the test in the source program to:

```
     if numeric-field = " abc"
```

and resubmit the source program.

Using the Correct Calling Convention

We recommend that to use the RM/COBOL library routines provided in Visual COBOL, you explicitly set the 1024 call-convention in the CALL statement.

```
program-id. Program1.
Special-Names.
call-convention 1024 is RM.
```

```
...

procedure division.

call RM "SYSTEM" using "cmd.exe /c mkdir sys02".

goback.
end program Program1.
```

RM/COBOL Library Routines

The following RM/COBOL routines are available in this COBOL system.

C$Century

Updates your COBOL programs to handle the year 2000 issue.

Note: When calling this routine, ensure you are using the 1024 calling convention.

This library routine retrieves the first two digits of the current year.

Syntax:

```
CALL "C$Century" USING value-buffer
```

Parameters:

value-buffer

A two-byte data item with a format of either unsigned numeric display (NSU) or alphanumeric display (ANS).

On Exit:

value-buffer	The first two digits of the current year.

Comments:

You can achieve the same result using the standard COBOL command `ACCEPT data-name FROM DATE YYYYMMDD` and then referencing the data name.

C$ConvertAnsiToOem

Converts a buffer containing ANSI characters to a buffer containing the corresponding OEM characters.

When calling this routine, ensure you are using the 1024 calling convention.

This is supported on Windows only.

Syntax:

```
CALL "C$ConvertAnsiToOem" USING ansi-buffer, oem-buffer
[, char-count]
```

Parameters:

ansi-buffer

PIC X(n)

oem-buffer

PIC X(n)

char-count

PIC 9(n)

On Entry:

ansi-buffer	The ANSI characters to be converted to OEM characters.
char-count	The number of characters to be converted.

Note: If omitted or if the value is invalid, the actual size of the shorter of ansi-buffer and oem-buffer is used.

On Exit:

oem-buffer	The converted OEM characters.

C$ConvertOemToAnsi

Converts a buffer containing OEM characters to a buffer containing the corresponding ANSI characters.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$ConvertOemToAnsi" USING oem-buffer, ansi-buffer
[, char-count]
```

Parameters:

oem-buffer

PIC X(n)

ansi-buffer

PIC X(n)

char-count

PIC 9(n)

On Entry:

oem-buffer	The OEM characters to be converted to ANSI characters.
char-count	The number of characters to be converted.

Note: If omitted or if the value is invalid, the actual size of the shorter of ansi-buffer and oem-buffer is used.

On Exit:

ansi-buffer	The converted ANSI characters.

C$DARG

Returns information about a parameter passed in the USING or GIVING phrases of the CALL statement that called a subprogram.

Restriction: This routine is supported in native COBOL only.

When calling this routine, ensure you are using the 1024 calling convention.

This information identifies the type and length of the argument and, when the argument is numeric or numeric edited, the number of digits and scale factor for the argument.

Syntax:

```
CALL "C$DARG" USING argument-number, argument-description
```

Parameters:

argument-number

pic 9(n)

argument-description

```
01 ARGUMENT-DESCRIPTION.
    02 ARGUMENT-TYPE PIC 9(2) BINARY(2).
    02 ARGUMENT-LENGTH PIC 9(8) BINARY(4).
    02 ARGUMENT-DIGIT-COUNT PIC 9(2) BINARY(2).
    02 ARGUMENT-SCALE PIC S9(2) BINARY(2).
    02 ARGUMENT-POINTER POINTER.
    02 ARGUMENT-PICTURE POINTER.
```

On Entry:

argument-number The ordinal position of the argument in the USING phrase of the CALL statement. The value zero returns the description of the GIVING phrase of the CALL statement.

On Exit:

argument-description Details of the passed parameter. Those details include:

argument-type The type of data item; see the table in the *Comments* section.

argument-length The number of character positions of the data item.

argument-digit-count The number of digits defined in the PICTURE character-string for an argument that is a numeric or numeric edited data item as indicated by the ARGUMENT-TYPE field value; otherwise, the value zero is returned for nonnumeric data items. The digit count for a numeric or numeric edited data item does not include any positions defined by the PICTURE symbol P, which represents a scaling position.

argument-scale The power of 10 scale factor (that is, the position of the implied or actual decimal point) for an argument that is a numeric or numeric edited data item as indicated by the ARGUMENT-TYPE field value; otherwise, the value zero is returned for nonnumeric data items. If the PICTURE symbol P was used in the description of the data item, the absolute value of the ARGUMENTSCALE value will exceed the ARGUMENT-DIGIT-COUNT value; in this case, a positive scale value indicates an integer with P scaling positions on the right of the PICTURE character-string and a negative scale value indicates a fraction with P scaling positions on the left of the PICTURE character-string

argument-pointer This parameter is not returned in this COBOL system.

argument-picture This parameter is not returned in this COBOL system.

Comments:

Use the C$NARG library routine to obtain the number of arguments passed in the CALL statement.

The actual number of arguments may exceed the number of formal arguments declared in the Procedure Division header of the program that calls C$DARG. All of the actual arguments can be accessed using C $DARG even though there is no formal argument name available for accessing the actual arguments beyond the number of formal arguments.

The following table is used to indicate the data type specified in the ARGUMENT TYPE field:

Type Number	RM/COBOL Data Type	Type Number	RM/COBOL Data Type
0	NSE	16	ANS
1	NSU	17	ANS (justified right)
2	NTS	18	ABS
3	NTC	19	ABS (justified right)
4	NLS	20	ANSE
5	NLC	21	ABSE
6	NCS	22	GRP (fixed length)
7	NCU	23	GRPV (variable length)
8	NPP	25	PTR
9	NPS	26	NBSN
10	NPU	27	NBUN
11	NBS	32	OMITTED
12	NBU		

Restriction: Data type OMITTED (type number 32) is not supported in this COBOL system.

C$Delay
Relinquishes the CPU for a length of time specified in seconds.

Note: When calling this routine, ensure you are using the 1024 calling convention.

This library routine allows other programs to run while the current program waits.

Syntax:

```
CALL "C$Delay" USING seconds
```

Parameters:
seconds

PIC 9(n)v999, where n is a digit from 1 to 7

On Entry:

seconds	The number of seconds.

Comments:
The amount of delay is not exact. It depends upon the particular machine configuration and the load on the machine.

C$GetEnv
Returns the value of an environment variable.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$GetEnv" USING name, value [, return]
```

Parameters:

name

PIC X(n)

value

PIC X (n)

return

PIC 9(n) BINARY, where n can be a digit from 1 to 9

On Entry:

name	The name of the environment variable.

On Exit:

value	The value of the environment variable, returned from the call.
return	The result code returned from the call: zero for success and non-zero for failure.

Comments:

On UNIX, environment variable names are case-sensitive. On Windows, environment variable names are not case-sensitive.

C$GetLastFileName

Retrieves the last filename used in a COBOL I/O statement (including OPEN and CLOSE).

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$GetLastFileName" USING filename
```

Parameters:

filename

PIC X(30)

On Exit:

filename	The name of the filename used in the last I/O operation.

Comments:

For REWRITE and WRITE statements, the COBOL filename associated with the specified file record-name is provided.

If the filename is longer than 30 characters, it is truncated to the right.

C$GetLastFileOp

Retrieves the last COBOL I/O operation performed.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Use this library routine within a declarative procedure after an I/O error has occurred.

Syntax:

```
CALL "C$GetLastFileOp" USING operation
```

Parameters:

operation

PIC X(20)

On Exit:

operation The name of the last I/O operation performed. The valid operations returned are:

Close	ReadRandom
CloseUnit	Rewrite
Delete	RewriteRandom
DeleteFile	Start
DeleteRandom	Unlock
Open	Write
ReadNext	WriteRandom
ReadPrevious	

Comments:

If the operation is longer than 20 characters, it is truncated to the right.

If the value SPACES is returned that indicates that no operation is available.

C$GetNativeCharset

Retrieves information about the native character set in effect for the current run unit.

Note: When calling this routine, ensure you are using the 1024 calling convention.

The native character set specifies how non-numeric data is encoded in memory and on data files.

Syntax:

```
CALL "C$GetNativeCharset" USING charset-name [, codepage-number]
```

Parameters:

charset-name

PIC X(n)

codepage-number

PIC 9(n)

On Exit:

charset-name The name of the character set in use for the current run unit after the call.

Note: For Windows, the name will have a value of “ANSI” or “OEM”. On UNIX, the value will be “NONE”.

codepage-number	The codepage number of the character set in use for the current run unit after the call.

Note: For Windows, the codepage number will be the system ANSI codepage number if *charset-name* contains “ANSI” and will be the system OEM codepage number if *charset-name* contains “OEM”. On UNIX, the value will be 0.

Comments:

The native character set for a run unit on Windows can be either ANSI or OEM.

The native character set for a run unit on UNIX is determined by the locale settings for the system.

C$LogicalAnd

Performs a bitwise logical AND operation on two or more non-numeric or numeric operands.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalAnd"
[GIVING result]
 USING operand1 {operand2} ...
```

Parameters:

result

PIC 9(n)

operand1

A non-numeric or numeric operand

operand2, 3, etc

A non-numeric or numeric operand that must be of the same data type as *operand1*

Note: If any non-numeric *operand2* is shorter than *operand1*, it is assumed to be padded on the right with binary zeroes.

On Entry:

operand1, 2, 3, etc	Non-numeric or numeric operands, which must be of the same data type as *operand1*.

On Exit:

result	The result of the operation or *operand1*.

Comments:

For non-numeric USING operands, the bitwise logical AND of all the operands replaces the value of *operand1*. The value of *result* is set to a non-zero value if any character of *operand1* is non-zero after the operation completes and zero otherwise.

For numeric USING operands, each operand is converted, if necessary, to a 32-bit binary integer. These 32-bit binary values are logically ANDed together. If the GIVING phrase is specified, the result of this operation is stored in *result* and the value of *operand1* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand1*.

C$LogicalComplement

Performs a bitwise logical One's Complement operation on a non-numeric or numeric operand.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalComplement"
[GIVING result]
 USING operand
```

Parameters:

result

PIC 9(n)

operand

A non-numeric or numeric operand

On Entry:

operand A non-numeric or numeric operand.

On Exit:

result The result of the operation or *operand*.

Comments:

If *operand* refers to a non-numeric data item, the bitwise logical One's Complement of the value of *operand* replaces the value of *operand*. The value of *result* is set to a non-zero value if any character of *operand* is non-zero after the operation completes and zero otherwise.

If *operand* refers to a numeric data item, the operand is converted, if necessary, to a 32-bit binary integer. The 32-bit binary value is logically One's Complemented. If the GIVING phrase is specified, the result of this operation is stored in *result* and the value of *operand* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand*.

C$LogicalOr

Performs a bitwise logical OR operation on two or more non-numeric or numeric operands.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalOr"
    [GIVING result]
     USING operand1 {operand2} ...
```

Parameters:

result

PIC 9(n)

operand1

A non-numeric or numeric operand

operand2, 3, etc

A non-numeric or numeric operand that must be of the same data type as *operand1*

Note: If any non-numeric *operand2* is shorter than *operand1*, it is assumed to be padded on the right with binary zeroes.

On Entry:

operand1, 2, 3, etc	Non-numeric or numeric operands, which must be of the same data type as *operand1*.

On Exit:

result	The result of the operation or *operand1*.

Comments:

For non-numeric USING operands, the bitwise logical inclusive OR of all the operands replaces the value of *operand1*. The value of *result* is set to a non-zero value if any character of *operand1* is non-zero after the operation completes and zero otherwise.

For numeric USING operands, each operand is converted, if necessary, to a 32-bit binary integer. These 32-bit binary values are logically inclusive OR'd together. If the GIVING phrase is specified, the result of this operation is stored in *result* and the value of *operand1* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand1*.

C$LogicalXor

Performs a bitwise logical exclusive OR operation on two or more non-numeric or numeric operands.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalXor"
    [GIVING result]
     USING operand1 {operand2} ...
```

Parameters:

result

PIC 9(n)

operand1

A non-numeric or numeric operand

operand2, 3, etc

A non-numeric or numeric operand that must be of the same data type as *operand1*

Note: If any non-numeric *operand2* is shorter than *operand1*, it is assumed to be padded on the right with binary zeroes.

On Entry:

operand1, 2, 3, etc	Non-numeric or numeric operands, which must be of the same data type as *operand1*.

On Exit:

result	The result of the operation or *operand1*.

Comments:

For non-numeric USING operands, the bitwise logical exclusive OR of all the operands replaces the value of *operand1*. The value of result is set to a non-zero value if any character of *operand1* is non-zero after the operation completes and zero otherwise.

For numeric USING operands, each operand is converted, if necessary, to a 32-bit binary integer. These 32-bit binary values are logically exclusive OR'd together. If the GIVING phrase is specified, the result of

this operation is stored in result and the value of *operand1* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand1*.

C$LogicalShiftLeft

Performs a logical shift left operation on a non-numeric or numeric operand.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalShiftLeft"
[GIVING result]
 USING operand [shiftcount]
```

Parameters:

result

PIC 9(n)

operand

A non-numeric or numeric operand

shiftcount

PIC 9(n)

On Entry:

operand	A non-numeric or numeric operand.
shiftcount	The number of positions to shift during the operation.

On Exit:

result	The result of the operation.

Comments:

If *operand* refers to a non-numeric data item, the value of the data item is shifted left by the number of bit positions specified by *shiftcount*. Any bits shifted off the left end are lost and zero-valued bits are shifted into the right end. The value of *result* is set to a non-zero value if any character of *operand* is non-zero after the operation completes and zero otherwise.

If *operand* refers to a numeric data item, the operand is converted, if necessary, to a 32-bit binary integer. The 32-bit binary value is logically shifted left by the number of bit positions specified by *shiftcount*. If the GIVING phrase is specified, the result of this operation is stored in *result* and the value of *operand* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand*.

C$LogicalShiftRight

Performs a logical shift right operation on a non-numeric or numeric operand.

When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$LogicalShiftRight"
[GIVING result]
 USING operand [shiftcount]
```

Parameters:

result

PIC 9(n)

operand

A non-numeric or numeric operand

shiftcount

PIC 9(n)

On Entry:

operand	A non-numeric or numeric operand.
shiftcount	The number of positions to shift during the operation.

On Exit:

result	The result of the operation.

Comments:

If *operand*refers to a non-numeric data item, the value of the data item is shifted right by the number of bit positions specified by *shiftcount*. Any bits shifted off the right end are lost and zero-valued bits are shifted into the left end. The value of Result is set to a non-zero value if any character of *operand* is non-zero after the operation completes and zero otherwise.

If *operand* refers to a numeric data item, the operand is converted, if necessary, to a 32-bit binary integer. The 32-bit binary value is logically shifted right by the number of bit positions specified by *shiftcount*. If the GIVING phrase is specified, the result of this operation is stored in *result* and the value of *operand* is not modified. If the GIVING phrase is not specified, the result of this operation is stored in *operand*.

C$NARG

Returns the number of parameters passed in the USING phrase of a CALL statement to the subprogram that contains the call to C$NARG.

Arguments specified explicitly as OMITTED in the USING list of the CALL statement are included in the count. The GIVING argument is not included in the count.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "C$NARG" USING parameter-count
```

Parameters:

parameter-count

PIC 9(3) BINARY, COMP-4 or COMP-1

On Exit:

parameter-count	The number of parameters passed.

C$SetEnv

Sets or clears the value of an environment variable.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Setting the value of an environment variable with C$SetEnv updates the corresponding environment variable immediately in the process space of the current run unit. Thus, when the RM/COBOL runtime system uses environment variables for such actions as file access name resolution, the call to C$SetEnv will have an immediate effect on that run unit.

Syntax:

```
CALL "C$SetEnv" USING name, value [, return]
```

Parameters:

name

PIC X(n)

value

PIC X(n)

return

PIC 9(n) BINARY, where n can be a digit from 1 to 9

On Entry:

name The name of the environment variable to set or clear.

value The value to which the environment variable is set. A value of SPACES indicates that the environment variable should be deleted.

On Exit:

return The result code returned from the call: zero for success and non-zero for failure.

Comments:

On UNIX, environment variable names are case-sensitive. On Windows, environment variable names are not case-sensitive.

C$RERR

Returns the expanded I/O completion status, based on an error code received at run-time.

This routine returns either a four-character or an eleven-character extended status code, depending upon the length of the data item specified in the USING phrase. This status is for the last attempted I/O operation. The value returned conforms to ANSI COBOL 1985.

Syntax:

```
CALL "C$RERR" USING extended-status
```

Parameters:

extended-status

PIC X(4) or PIC X(11)

On Exit:

extended-status The data item into which the expanded I/O completion status is stored in ASCII characters.

Comments:

If *extended-status* is four characters in length, the first two character positions contain the same digits as would the file status data item on completion of the I/O operation. The last two character positions provide additional information about the file status. In cases where only two digits for a status are shown, the last two character positions will contain ASCII zeroes. Although most statuses contain only the decimal digits 0 to 9, note that the hexadecimal digits A to F are possible in some character positions. Refer to *Appendix A: Runtime Messages* of the *RM/COBOL User's Guide* for a full list of status codes.

If *extended-status* is eleven characters in length, the first two character positions (positions one and two) contain the same digits as would the file status data item on completion of the I/O operation. In cases where Appendix A shows only two digits for a status, the remaining nine character positions contain ASCII blanks. In cases where Appendix A shows four digits for a status, character position three contains an ASCII comma, character positions four and five contain the last two digits of the status, and the remaining six character positions contain ASCII blanks. For permanent errors, that is, when the first two digits are 30, character position three contains an ASCII comma, character positions four and five contain a two-digit OS code (see the table below), character position six contains an ASCII comma, and character positions seven through eleven contain a five-digit, OS-specific error code. Refer to the *Input/Output Errors* section of the *RM/COBOL User's Guide*.

Table 1: The two-digit OS codes

Code	Description
00	Unknown OS error
01	File Manager Detected error
04	UNIX error
06	Btrieve error
10	Open File Manager error
11	C Library error
12	MS-Windows error
15	RM/InfoExpress Server error
16	RM/InfoExpress Client error
21	RM/InfoExpress WinSock error

DELETE
Deletes a file.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "DELETE" USING file-name [exit-code]
```

Parameters:
file-name
PIC X(n)

exit-code
PIC S9(4) BINARY

On Entry:

file-name The full or relative pathname of the file to be deleted.

On Exit:

exit-code The exit code of the command upon return from the operating system: zero for success and non-zero for failure.

Comments:

The values for the old-name parameter may be quoted with double quotes (") or single quotes ('). When the name is quoted, the quotes are removed, but the name is not otherwise modified. If the name is not quoted, the first control character terminates the name on Windows and the first white space character terminates the name on UNIX. On Windows, trailing spaces are removed from unquoted names.

The old-name data item must be less than 1024 characters in length.

RENAME
Renames a file.

Note: When calling this routine, ensure you are using the 1024 calling convention.

Syntax:

```
CALL "RENAME" USING old-name new-name [exit-code]
```

Parameters:
old-name
PIC X(n)

new-name
PIC X(n)

exit-code
PIC S9(4) BINARY

On Entry:

old-name	The source filename.
new-name	The target filename.

On Exit:

exit-code	The exit code of the command upon return from the operating system: zero for success and non-zero for failure.

Comments:

The values for the old-name and new-name parameters may be quoted with double quotes (") or single quotes ('). When the name is quoted, the quotes are removed, but the name is not otherwise modified. If the name is not quoted, the first control character terminates the name on Windows and the first white space character terminates the name on UNIX. On Windows, trailing spaces are removed from unquoted names.

The old- and new-name data item must be less than 1024 characters in length.

SYSTEM
Executes an arbitrary operating system command.

Syntax:

```
CALL "SYSTEM" USING command-line [repaint-screen] [exit-code]
```

Parameters:
command-line
PIC X(n)

repaint-screen

This parameter is ignored in this COBOL system.

exit-code

This parameter is ignored in this COBOL system.

On Entry:

command-line	An alphanumeric data item that contains the command line to be passed to the operating system.
repaint-screen	This parameter is ignored in this COBOL system.

On Exit:

exit-code	This parameter is ignored in this COBOL system.

Comments:

The implementation of this library routine is identical to the existing Micro Focus version of SYSTEM.

RM/COBOL File Handling

When you migrate your RM/COBOL applications to Visual COBOL, you can continue to use the same data files.

Alternatively, you can use a data migration tool to convert an RM/COBOL data file to Micro Focus format. The data migration tool, `RMMFDataMigration.exe`, is available from the folder in which the product samples are installed.

Configuring Access to RM/COBOL Indexed Data Files

To handle RM/COBOL indexed data files, you map a file to IDXFORMAT=21 in the File Handler configuration file.

Within the configuration file, you can apply IDXFORMAT 21 to all files in a particular folder, all files with a specific file extension, or a single file. See *Format of the Configuration File* for the tags that you can use for the mapping, and the order in which settings in these tags are applied.

The order that the mapping is applied is important, as conflicting settings can be overwritten; for example, the following excerpt of the configuration file sets all files in `c:\files\rmfiles` to IDXFORMAT 21 and all files with a `.DAT` extension to IDXFORMAT 17:

```
[FOLDER:C:\\files\\rmfiles]
IDXFORMAT=21

[*.DAT]
IDXFORMAT=17
```

If there is a `.DAT` file in `c:\files\rmfiles`, the mappings are applied according to the type of tag. In the case above, mappings in the extension tag are applied after mappings in the FOLDER tag, and so the `.DAT` file in that directory has an IDXFORMAT of 17.

By default, the File Handler handles all sequential and relative data files, but if you want to handle them through the RM/COBOL file handler, use the INTEROP=RM configuration option; however, in cases where the INTEROP and IDXFORMAT mappings conflict, the INTEROP setting will override IDXFORMAT for your RM/COBOL indexed data files.

RM/COBOL File Status Codes

RM/COBOL file status codes take a 2-digit form in the file status variable, by combining the values of the Status Key 1 and 2 columns, or a 4-character or 11-character extended file status code, which can be retrieved using the C$RERR standard library routine.

To always return RM/COBOL file status codes:

- Set environment variable COBFSTATCONV to the RM/COBOL setting:

```
set COBFSTATCONV=rmstat
```

- Set the COBFSTATCONV Compiler directive.

Status Key 1	Status Key 2	Extended File Status Code	Description
3	5	9/013	File not found.
3	5	9/188	Filename too large.
3	7	9/035	Incorrect access permission.
3	7	9/037	File access denied.
3	8	9/138	File is closed with lock - cannot open.
3	8	9/210	File is closed with lock.
4	1	9/141	File already open - cannot be opened.
4	2	9/142	File not open - cannot be closed.
4	3	9/143	REWRITE/DELETE not after successful READ
4	6	9/146	No current record defined for sequential read.
4	7	9/147	Wrong open or access mode for READ/ START.
4	8	9/148	Wrong open or access mode for WRITE.
4	9	9/149	Wrong open or access mode for REWRITE/ DELETE.
9	3	9/065	File locked.
9	8	9/071	Bad indexed file format.
9	8	9/139	Record length or key inconsistent.
9	9	9/068	Record is locked.

Enabling CTF to Trace RM/COBOL Data Files

Enable the Micro Focus Consolidated Tracing Facility (CTF) to trace activity with your RM/COBOL data files.

To enable CTF:

- Set the following environment variables:

```
set MFTRACE_CONFIG=ctf.cfg
set MFTRACE_LOGS=pathname   *> if not set, logs are stored in the current
folder.
```

- In ctf.cfg, set the following:

```
mftrace.dest                = textfile
mftrace.level.mf.rts        = info
mftrace.comp.mf.rts#eprintf = true
```

- Set the following environment variable:

```
set A_CONFIG=rmfm.cfg    *> rmfm is your RMFM configuration file
```

- In rmfm.cfg, set the following:

```
DEFAULT_FILESYSTEM RMFM
FILE_TRACE 3   *> values 0-9 set amount of activity traced.
```

When you run your program, a log-file is produced that includes the activity with the RM/COBOL data files.

For more information on CTF, see *Introduction to the Consolidated Tracing Facility*.

Compatibility with XML Extensions

Note: This functionality is supported in native COBOL only.

XML Extensions has many capabilities. The major features support the ability to import and export XML documents to and from COBOL working storage. Specifically, XML Extensions allows data to be imported from an XML document by converting data elements (as necessary) and storing the results into a matching COBOL data structure. Similarly, data is exported from a COBOL data structure by converting the COBOL data elements (as necessary) and storing the results in an XML document.

For more information about XML Extensions, refer to the *XML Extensions User's Guide*, available from the RM/COBOL product documentation set, in the SupportLine section of the Micro Focus Web site.

For RM/COBOL users that utilize XML Extensions, here is a summary of compatibility issues that you need to be aware of when working in this COBOL system. Refer to this list and the *RM/COBOL Conversion Issues* list in the *Compatibility with RM/COBOL* section

Click a summary title for a fuller explanation and workaround, where possible.

Additional Parameter Required with XML Extensions Processing Statements

In statements that use a Document Pointer parameter, you are also required to pass an additional Document Length parameter.

When using XML Extensions processing statements, each Document Pointer parameter must be immediately followed by a Document Length parameter. This applies to the following statements:

- XML EXPORT TEXT
- XML IMPORT TEXT
- XML TEST WELLFORMED-TEXT
- XML VALIDATE TEXT
- XML GET TEXT
- XML PUT TEXT
- XML TRANSFORM TEXT

Note: XML FREE TEXT does not require that you use the Document Length parameter.

Solution:

Ensure that the Document Length parameter (MY-DOCUMENT-LENGTH) is specified immediately following the Document Pointer parameter (MY-DOCUMENT-POINTER) when calling an XML Extensions processing statement:

When the statement is outputting data, the statement will set MY-DOCUMENT-LENGTH:

```
XML EXPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    MY-DOCUMENT-LENGTH
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

When the statement is inputting data, you must set MY-DOCUMENT-LENGTH before the statement is processed:

```
XML IMPORT TEXT
    MY-DATA-ITEM
    MY-DOCUMENT-POINTER
    MY-DOCUMENT-LENGTH            *> The size of the data item pointed to by
MY-DOCUMENT-POINTER.
    "MY-MODEL-FILE".
IF NOT XML-OK GO TO Z.
```

COBOL programs using BIS
Programs in this COBOL system that are used with the Xcentrisity Business Information Server (BIS) must end with the GOBACK statement, not the STOP RUN statement. Also, messages for the BIS trace log must be generated by calling the B$Trace library program, not the DISPLAY statement.

Programs that are used with the BIS must not use the STOP RUN statement, as this will terminate the MF run-time prematurely and the BIS will be unable to process any further web service requests.

In RM/COBOL, programs that are used with BIS capture the output of a DISPLAY statement and place it in the BIS trace log. In this COBOL system, to place messages in the BIS trace log, use the B$Trace library routine.

Solutions:

To ensure that programs used with the BIS do not prematurely terminate the MF run-time, use the GOBACK statement in those programs.

To place messages in the BIS trace log, call the B$Trace library program, using the same identifiers or literals, but not figurative constants, that you would use in a DISPLAY statement.

Note: Numeric data items can be of any data type and are converted to a numeric string by B$Trace.

```
call "B$Trace" using "Log message: " MyMessage " " MyStatus.
```

Conflicts Between Model File-names and XML Data Files
In this COBOL system, model file-names, as created by the compiler, are of the form *program-name*.xml. You should ensure your XML data files do not share the same name as this, to avoid any conflicts.

In this COBOL system, if the ModelFileName#DataName parameter does not include a hash, it is always treated as a model data-name, and the model file-name is assumed to be *program-name*.xml for the program (or one of its callers) that executed an XML Extensions export or import statement. With this in mind, if you do not explicitly set a model file-name, you should ensure that your XML data files do not share the same name as your COBOL programs when performing import and export XML Extensions statements.

Solution:

To avoid conflicts between model file-names and XML data file-names, do one of the following:

- Ensure you set the DocumentName parameter in your import and export statements to a name other than your COBOL program name.

- If you want to keep your XML data file-names the same as the program-name, rename the model file-name after compilation and specify the new name in the value of the ModelFileDataName parameter before the hash, separating it from the ModelDataName.

When using the second technique, it is recommended that the compilation be done with a script that includes the renaming command, to avoid forgetting this step.

Notes:

In RM/COBOL, you can use the environment variable RM_MISSING_HASH to determine the meaning of the ModelFileDataName parameter when the hash is omitted. In this COBOL system, the environment variable is not supported.

Also, RM/COBOL v12 and later generally did not use model files because the model was embedded in the object program file; this COBOL system is more like RM/COBOL v11 and earlier, which always used model files. Thus, care must be taken to distribute model files with applications that use XML Extensions.

Creating an XML Model File

To create an XML model file for use with XML Extensions, compile your application with the XMLGEN Compiler directive.

By default, this creates an XML model file, named `output-name.xml`, that includes data descriptions from the File Section of your code. This file is created in the project's root directory.

Note: The output-name defaults to the project title, but you can change it in the project's properties.

There are two parameters to this directive that enable you to alter the default behavior:

To change the location of the model file, specify:

```
XMLGEN(pathname)
```

where *pathname* is the absolute or relative path name of the `.xml` file, which is prefixed to the default file name. You can also use an environment variable, which will resolve to a path name during runtime:

```
XMLGEN($myXML/)
```

where *myXML* is an environment variable set to the absolute or relative path name of the `.xml` file, which is prefixed to the default file name.

To include data descriptions from the Working-storage section in the model file, specify:

```
XMLGEN(ws)
```

See *Restricted data items with XML Extensions* for a workaround to include data descriptions from other Data Division sections of your source code.

When using both parameters in your program, ensure you set *XMLGEN(pathname)* before you set *XMLGEN(ws)*; otherwise, *XMLGEN(pathname)* suppresses the *XMLGEN(ws)* option, which results in only File Section data descriptions in the XML file.

Displaying the Status of XML Extensions Statements

In this COBOL system, use the XML-Status-Edited data item to display the status result of an XML Extensions statement execution.

In RM/COBOL, XML-Status, the data item used to display the status result of an XML Extensions statement execution, is defined as Display Usage. In this COBOL system, XML-Status is defined as:

```
03 XML-Status          PIC S9(4) COMP-5.
```

Therefore, an additional declaration is made in *lixmldef.cpy*, so that you can easily use the status result in your code:

```
03 XML-Status-Edited        PIC +9(4).
```

When an XML Extensions statement is executed, the value of XML-Status-Edited is not set, so you need move XML-Status to XML-Status-Edited before you can use the result.

Importing and Exporting Ambiguous Data-names

In RM/COBOL, if you attempt to export an ambiguous data-item to a model file, an error is produced. If you attempt to import to an ambiguous data-item, the data is placed in the first occurrence of the named data-item.

In this COBOL system, if you attempt to export an ambiguous data-item to a model file, a warning message is produced and the first occurrence of the named data-item is exported. Similarly, If you attempt to import to an ambiguous data-item, a warning message is displayed and the data is placed in the first occurrence of the named data-item.

Example:

```
01 Group01.
    02 GroupA.
      03 NumItem     PIC s9(5).
      03 StrItem     PIC X(5).
        02 GroupB.
          03 NumItem  PIC s9(5).
          03 StrItem  PIC X(5).

--------------------------------
<StrItem> ABCDE </StrItem>   *> this produces a warning and
updates StrItem in GroupA

<GroupB><StrItem> ABCDE </StrItem></GroupB>  *> this updates
StrItem in GroupB
```

Invalid Characters in Condition Names

In this COBOL system, if you use mark-up characters as values for condition names, this can produce invalid XML when exporting code using XML Extensions.

Mark-up characters, such as "<", ">" or "&" used in the values for condition names will produce invalid model files when using XML Extensions. The model files will cause parse errors when loaded by XML Extensions using the XML parser; XML Extensions will report the parse error and be unable to perform the requested export or import.

```
88 cond-name VALUE "<br/>".
```

Solution:

In this COBOL system, you must modify the COBOL source code, to eliminate mark-up characters in condition-name values.

Restricted data items with XML Extensions

In this COBOL system, you cannot use data items described in any section other than the File or Working Storage Sections, as model data names.

To export data items from or import XML data into this COBOL system, use the XMLGEN Compiler directive to create a model file, for use with XML Extensions.

The model data names specified in the model file are determined by XMLGEN:

- XMLGEN with no parameter specified produces model data names for data items/structures in the File Section only.
- XMLGEN(ws) produces model data names for data items/structures in the Working Storage Section only.

Important: Data items/structures described in the Linkage Section, Communication Section, Local-Storage Section and Thread-Local-Storage Section cannot be used as model data names in a model file.

Solution:

Using a copybook containing your data items, compile a dummy program that copies the descriptions into the Working Storage section, and then use the XMLGEN(ws) Compiler directive to create a model file containing the required data items.

Notes:

The data items used at runtime when the model file is used can be in any section of the data division.

Unable to Use Data Items Declared in Nested Programs
In this COBOL system, you cannot use data items declared in nested programs, as model data-names.

Solution:

Using a copybook containing your data items, compile a dummy program that copies the descriptions into the Working Storage section of your top-level program, and then use the XMLGEN(ws) Compiler directive to create a model file containing the required data items.

User-names Longer than 127 Bytes are Truncated
In RM/COBOL, you can specify user-names (data-names, procedure-names, program-names, etc) up to 240 characters long. In this COBOL system, user-names longer than 127 bytes in length are truncated and a warning message is produced.

Solution:

Results may be affected if the truncated user-name is used with XML Extensions, to export or import XML documents; therefore, we recommend that you keep user-names to 127 bytes or less.

Using the Correct Calling Convention

In this COBOL system, XML Extensions uses the standard COBOL calling convention. If your programs are also using the standard library routines implemented for RM/COBOL compatibility, you need to be aware that these are called using the 1024 calling convention.

Solution:

Generally, you should explicitly use the 1024 calling convention when calling your RM/COBOL standard library routines, but if you are using the DEFAULTCALLS Compiler directive to set this calling convention, you will need to override it when calling to XML Extensions.

Native COBOL compared with managed COBOL

Native COBOL and managed COBOL differ in how they compile and how the run-time management services, such as security, threading and memory management are provided.

Managed COBOL on the .NET platform compiles to Microsoft Intermediate Language (IL), and native COBOL compiles to machine code. Both managed and native COBOL can run on any Windows platform when compiled.

For .NET managed code, the management services are provided by the Microsoft Common Language Runtime (CLR). For native COBOL, the management services are available in the operating system, and your code has to call the appropriate services depending on the operating system. The management services enable seamless interoperation of COBOL programs with programs in other managed languages.

What Is .NET managed code?

.NET managed code compiles to Microsoft Intermediate Language (IL). The IL is stored in an assembly, along with meta data that describes the classes, methods, and attributes (such as security requirements) of the code you've created.

.NET managed code runs in the Microsoft Common Language Runtime (CLR). The CLR does Just In Time (JIT) compilation. That is, when you load an assembly, the CLR JITs the assembly code the first time it is executed. There is a small performance penalty as an application loads, but because the CLR compiles your code, it doesn't do it again (until next time you restart it).

The CLR is responsible for managing your application code at run time, and provides security, memory management and so on.

Building native and managed COBOL applications

You use the IDE to develop, compile and debug your applications, for both native and managed code. You can write new COBOL code or you can recompile existing COBOL as managed or native code, potentially without any code changes.

You can deploy and further debug the application under the run-time system provided by COBOL Server. .NET COBOL applications are deployed to Windows platforms running the .NET Framework.

Customer Feedback

We welcome your feedback regarding Micro Focus documentation.

Submit feedback regarding this Help

Click the above link to email your comments to Micro Focus.