

For Microsoft® Windows®

NetCOBOL

PowerCOBOL User's Guide

FUJITSU

Sixth Edition: October 2002

The contents of this manual may be revised without prior notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Fujitsu Limited.

© 1996-2002 Fujitsu Limited. All Rights Reserved.

Preface

Fujitsu PowerCOBOL (referred to as PowerCOBOL throughout this manual) is a graphical application development environment designed for COBOL programmers creating applications with Windows® graphical user interfaces (GUI). This manual:

- Introduces PowerCOBOL
- Explains the functions of the product
- Describes how to use it

Audience

This manual is intended for programmers using PowerCOBOL to develop Windows applications. We recommend that programmers also refer to the "PowerCOBOL Reference" and the "NetCOBOL Users Guide for Windows" for further information.

Organization

The table below shows how this manual is organized.

Chapter /Appendix	Description
Chapter 1. Introduction	Introduces PowerCOBOL and outlines the major functions of the product.
Chapter 2. The Development Environment	Describes the development environment, including the menus and options in the user interface.
Chapter 3. Developing Your First PowerCOBOL Application	Steps through the creation of a simple PowerCOBOL application.
Chapter 4. Creating and Editing an Application Window	Describes how to create and edit an application window (form) including how to set styles, create graphical user interface items, menu bars, etc.
Chapter 5. Writing Event Procedures	Describes how to write event procedures, including inserting properties and methods.
Chapter 6. Creating Executable Programs	Describes how to create and edit a project, compile, link and build a program.
Chapter 7. Debugging The Program	Describes how to use the debugger, set breakpoints, and perform other debugging functions.
Chapter 8. Developing Your First ODBC Application	Steps through the development of simple and complex ODBC applications using PowerCOBOL.
Chapter 9. PowerCOBOL Programming Techniques	Describes PowerCOBOL programming techniques, including how to create and call Dynamic Link Libraries, and how to configure PowerCOBOL applications.
Chapter 10. Using PowerCOBOL Methods and Properties	Describes how to use some of the methods and properties supplied with PowerCOBOL.
Chapter 11. Creating and Using Custom Controls	Discusses creating and using custom OLE and ActiveX controls in PowerCOBOL applications.

Conventions Used in This Manual

The following table lists the meaning of symbols appearing in this manual.

.	Any period shown must be written in the same position.
(period)	
setup	Characters you enter appear in bold.
The <i>sheet</i> acts as an application creation form.	Italics are occasionally used for emphasis.
PROCEDURE DIVISION	This font is used for examples of program code.
CHECK WITH PASCAL LINKAGE ALL PARAGRAPH-ID COBOL <u>ALL</u>	Commands, statements, clauses and options you enter or select appear in uppercase. Programs section names, and some proper names also appear in uppercase. Defaults are underlined.
{ABC DEF}	Indicates that one of the enclosed items delimited by is to be selected.
[def]	Indicates that the enclosed item may be omitted.
Edit, Literal	Names of pull down menus and options appear with the initial letter capitalized.
...	Ellipses indicate that the item immediately preceding can be specified repeatedly.
ENTER	Small capitol letters are used for the names of keys and key sequences such as ENTER and CTRL+R. A plus sign (+) indicates a combination of keys.
Special characters such as +, -, >, <, =, >=, <=, ->	Special characters are key words. They are not underlined, but must not be omitted.

Related Manuals

PowerCOBOL Reference

NetCOBOL Getting Started

NetCOBOL User's Guide for Windows

NetCOBOL Language Reference

Information on related topics, such as the DDE and OLE functions of Microsoft Windows operating system would be helpful.

Product Names and Trademarks

Abbreviations of the product names used in this manual are listed below.

Product Name	Abbreviation
Microsoft® Windows® 95 operating system	Windows 95
Microsoft® Windows® 98 operating system	Windows 98
Microsoft® Windows® Millennium Edition	Windows Me
Microsoft® Windows NT® Server Network operating system Version 4.0	Windows NT or Windows NT 4.0
Microsoft® Windows NT® Server Network operating system, Enterprise Edition Version 4.0	Windows NT or Windows NT 4.0
Microsoft® Windows® 2000 Server operating system	Windows 2000
Microsoft® Windows® 2000 Advanced Server operating system	Windows 2000
Microsoft® Windows® XP Professional operating system	Windows XP
Microsoft® Windows® XP Home Edition operating system	Windows XP

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

Fujitsu, NetCOBOL and Fujitsu PowerCOBOL are trademarks or registered trademarks of Fujitsu Limited.

Other product names are trademarks or registered trademarks of each company. Trademark indications are omitted for some system and product names described in this manual.

Contents

Chapter 1. Introduction	9
Overview	10
Make Use of the PowerCOBOL Sample Programs.....	22
 Chapter 2. The Development Environment.....	 19
The PowerCOBOL Project Manager	20
The PowerCOBOL Form Editor	41
The PowerCOBOL Build Facility	42
The PowerCOBOL Debugger	44
 Chapter 3. Developing Your First PowerCOBOL Application.....	 49
Overview	49
Enhancing the Application	65
Further Enhancing the Application.....	72
 Chapter 4. Creating and Editing an Application Window (Form).....	 83
Overview	84
Creating a Form	84
Setting the Form Properties.....	87
Creating a Control	90
Setting a Control's Initial Properties	93
Manipulating Controls	94
Setting Control Order.....	99
Placing Controls in an Array.....	102
Setting Fonts	106
Creating a Menu Bar.....	107
Creating Toolbars	112
Printing Forms.....	119
Printing Procedure Code.....	119

Chapter 5. Writing Event Procedures.....	121
Overview	121
Inserting a Control Name	128
Writing an Event Procedure.....	129
Inserting Properties and Methods	138
Searching and Replacing Text Strings.....	148
 Chapter 6. Creating Executable Programs	 159
Overview	159
Project Files	160
Installing Applications	169
Using Precompilers with PowerCOBOL.....	170
 Chapter 7. Debugging the Program	 173
Overview	173
The PowerCOBOL Debugging Environment	177
 Chapter 8. Developing Your First ODBC Application.....	 197
Overview	197
Beginning the Development Process.....	202
 Chapter 9. PowerCOBOL Programming Techniques	 225
Traditional COBOL programming methodology versus PowerCOBOL's event-driven programming methodology	226
PowerCOBOL Application Execution Flow	228
Sharing Data between PowerCOBOL Forms and Procedures	230
Calling COBOL DLLs.....	233
PowerCOBOL Overall Application Architecture	234
Working with Multiple Windows in a PowerCOBOL Application	237
Working with Objects.....	239
Receiving Object References in Event Procedures.....	242
Opening a PowerCOBOL Form from COBOL	244
Calling the Windows API from PowerCOBOL	246
Sharing Data between a COBOL Application Calling a PowerCOBOL Application.....	248
Working with VT_BSTR and VT_VARIANT Data Types	250
Notes on Programs that Include COBOL and PowerCOBOL Procedures.....	251

Chapter 10. Using PowerCOBOL Methods and Properties	253
Finding Method Information	254
Using the Supplied Methods	264
 Chapter 11. Creating and Using Custom Controls	 267
Developing the Initial Clock Control.....	269
Enhancing the Clock Control.....	281
Using Other 3 rd Party Controls Within PowerCOBOL	287
Using ActiveX in HTML	288
The OLE Control.....	289
 Index.....	 291

Chapter 1. Introduction

This chapter introduces Fujitsu's PowerCOBOL product and outlines its major functions including:

- What you can do with PowerCOBOL
- A brief look at PowerCOBOL application development

Overview

PowerCOBOL is a graphical application development tool for COBOL programmers. The PowerCOBOL development environment allows programmers to use their existing COBOL knowledge to efficiently build and execute complex graphical user interface (GUI) applications in the Microsoft Windows environment. PowerCOBOL simplifies the process of programming for event-driven object-oriented applications and abstracts Windows APIs to a higher level.

PowerCOBOL enables programmers to perform all of the steps associated with developing complex, graphical, client/server, Windows applications.

With PowerCOBOL, you can:

- Develop graphical user interface (GUI) applications
- Include event-driven programming objects
- Enhance applications with standard COBOL syntax
- Use ODBC and ADO to access a wide variety of Database systems both locally and remotely
- Develop multimedia applications
- Develop client/server applications
- Develop applications that interact with other open object environments
- Create OLE, ActiveX and COM components developed in other languages

The following is a brief look at some of these tasks.

Developing Graphical User Interface (GUI) Applications

PowerCOBOL provides you with a main application development facility, the Project Manager, to create and manage your development projects.

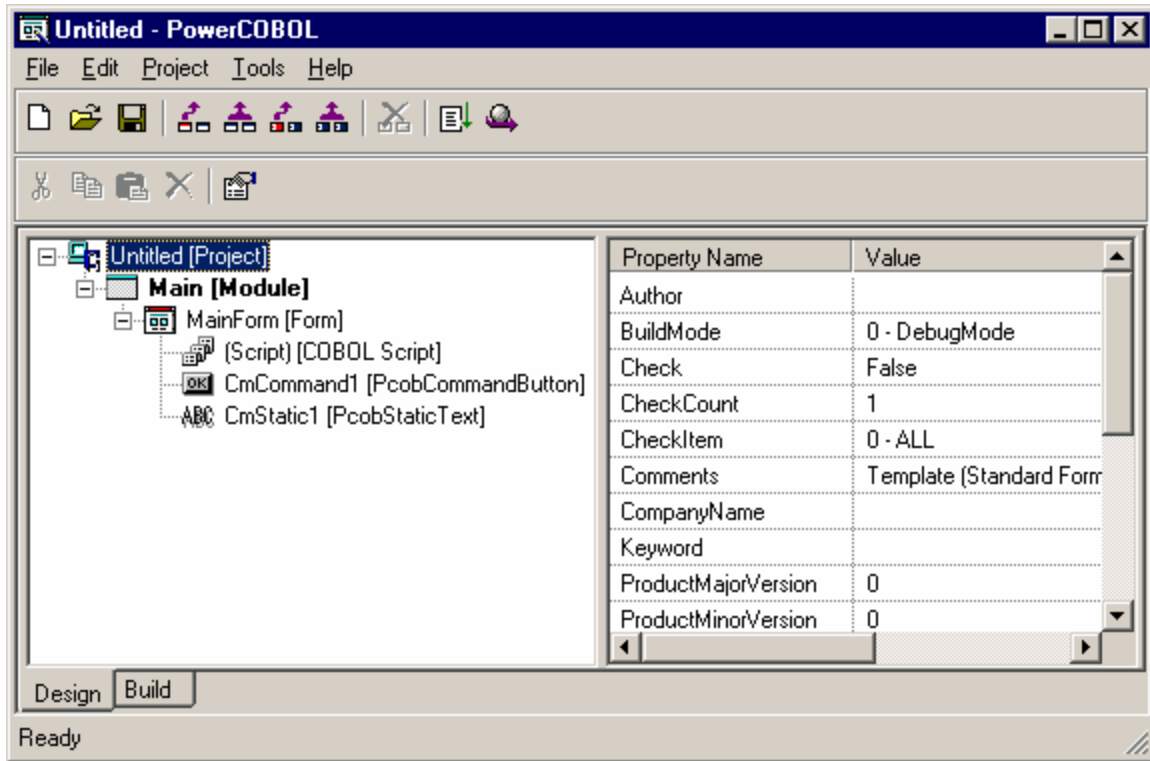


Figure 1.1. The PowerCOBOL Project Manager

The Project Manager includes wizards to automatically generate application templates, whenever you choose to create a new project.

PowerCOBOL also includes a sophisticated GUI form (window) painter and editor (the Form Editor) to simplify the development of the graphical user interface.

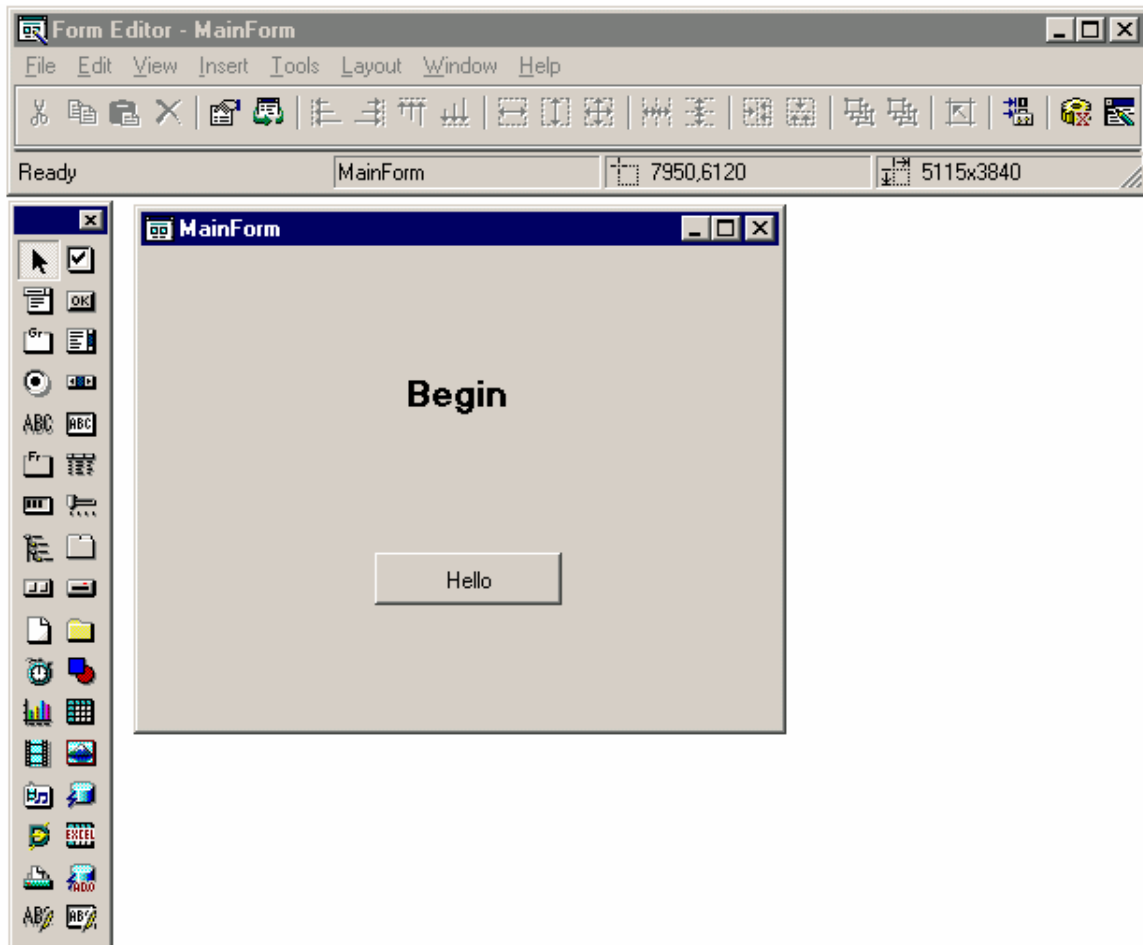


Figure 1.2. The PowerCOBOL GUI Form Editor

Graphical controls are easily dragged and dropped from the Toolbox palette onto a form (window). Event procedures may be developed for any application control event, and PowerCOBOL handles the management of the application code.

Components available directly from the Toolbox palette include:



- CheckBox control



- ComboBox control



- CommandButton control



- GroupBox control



- ListBox control



- OptionButton control



- ScrollBar control



- StaticText control



- TextBox control



- Frame Control



- ListView control



- ProgressIndicator control



- Slider control



- TreeView control



- Tab control



- Toolbar Control



- DriveList control



- FileList control



- FolderList control



- Timer control



- Shape control



- Graph control












- Table control



- Animation control



- Image control

-  - MCI (Media Control Interface) control
-  - DB (Database) Access control
-  - DDE (Dynamic Data Exchange) control
-  - Excel Connection control
-  - Print control
-  - Label control (Introduced in Version 7 – if you installed Version 7 on a machine that had an earlier version of PowerCOBOL this control must be added to the Toolbox palette)
-  - Edit control (Introduced in Version 7 – if you installed Version 7 on a machine that had an earlier version of PowerCOBOL this control must be added to the Toolbox palette)
-  - ADO Data Control (Introduced in Version 6.1 - if you installed Version 6.1 or 7 on a machine that had an earlier version of PowerCOBOL this control must be added to the Toolbox palette)
-  - Microsoft Transaction Server Support Control - (Introduced in Version 6.1 - must be added to the Toolbox palette)

Adding Controls to the Toolbox Palette

In addition to the controls provided in the Toolbox palette, you may insert additional custom controls such as the Fujitsu PowerCOBOL ADO Data and Microsoft Transaction Server Support controls or external controls created in other development environments such as Microsoft's Visual Basic.

You access these other controls by selecting Custom Controls from the Tools menu in the Form Editor. When you select this option, PowerCOBOL will scan your current system and find all registered custom controls and then dynamically build a list for you to select from.

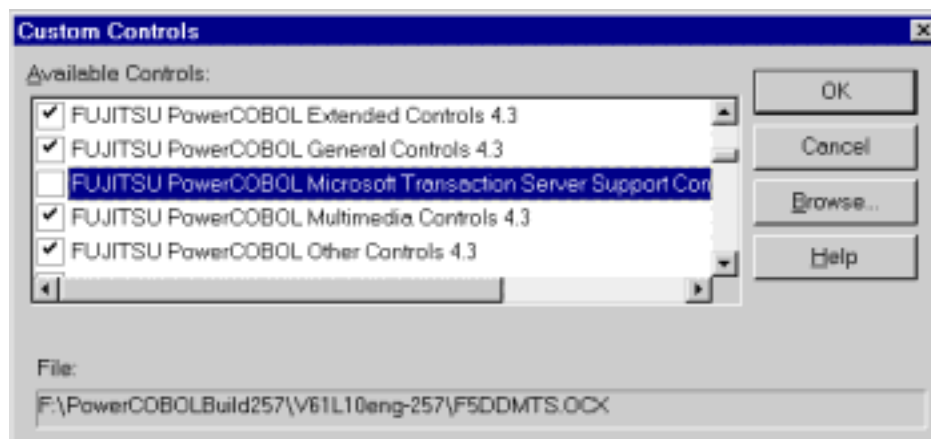


Figure 1.3. The Custom Controls dialog box

Including Event-driven Programming Objects

Windows applications rely on *events*. Events are the expected results of simple user actions. For example, at the click of a button, users expect sub-windows to open with choices from menus, check boxes, or list boxes.

Dragging the mouse across a certain portion of the screen will generate an event stating the mouse has been dragged over this area.

Based on the cursor position in the window, another click may be expected to cause colors to change, the window to close, or one of many other Windows events.

Programmers using a traditional, third-generation programming language have to write code to initiate such events. However, in PowerCOBOL the support logic for event-driven programming (standard Windows events) is built-in, as shown in the following figure, to allow programmers to focus their attention on other tasks.

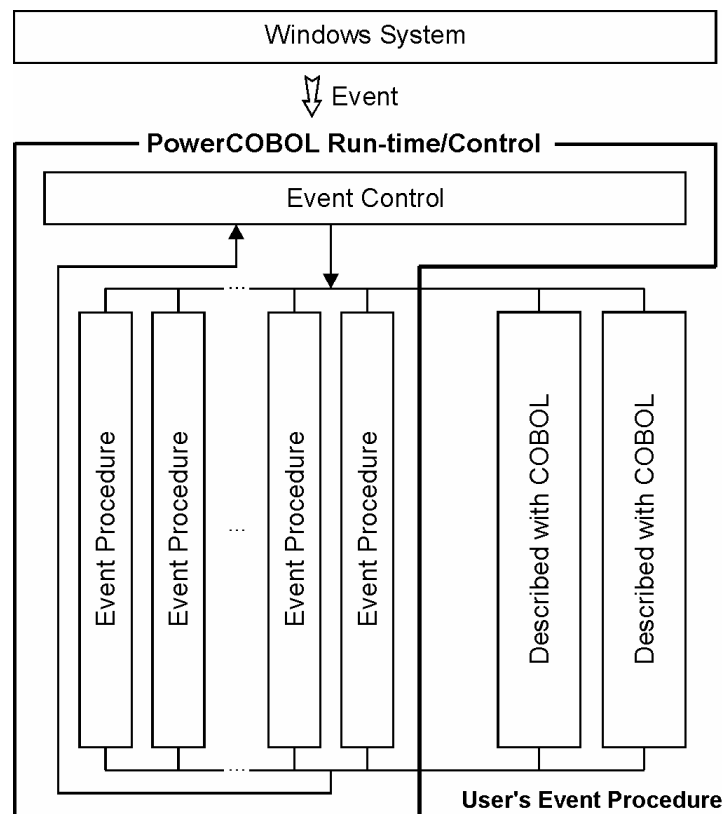


Figure 1.4. Event-driven programming

As you may be able to deduce from Figure 1.4, PowerCOBOL vastly simplifies the task of writing complex GUI applications. Instead of being forced to write a large and complex application, your development task is easily broken down into writing individual event procedures.

PowerCOBOL puts all of these event procedures together properly, thus relieving you of a great deal of application design and management.

Enhancing Applications with Standard COBOL Syntax

PowerCOBOL supports COBOL language functions to use programmers' existing skills. Event procedures are coded using the native COBOL language.

Beginning with the version 5.0 product, you can optionally choose OO COBOL as the scripting language.

PowerCOBOL provides extensions to the COBOL language to handle application objects, events and their properties. These extensions have been designed with COBOL in mind (in the form of a simplified scripting language). The scripting language itself looks like COBOL. It makes use of standard COBOL verbs such as MOVE to set properties, and enforces a standard COBOL syntax.

The following figure illustrates an example event procedure for the simple Hello World! application form (window) shown in Figure 1.2.

Please note that many of the coding examples showing the PowerCOBOL Editor have the status bar and the toolbar turned off, so your Editor will appear differently if you have either of these enabled. You can enable or disable the status bar and/or the toolbar in the PowerCOBOL Editor in the Options dialog box found under the Tool menu.

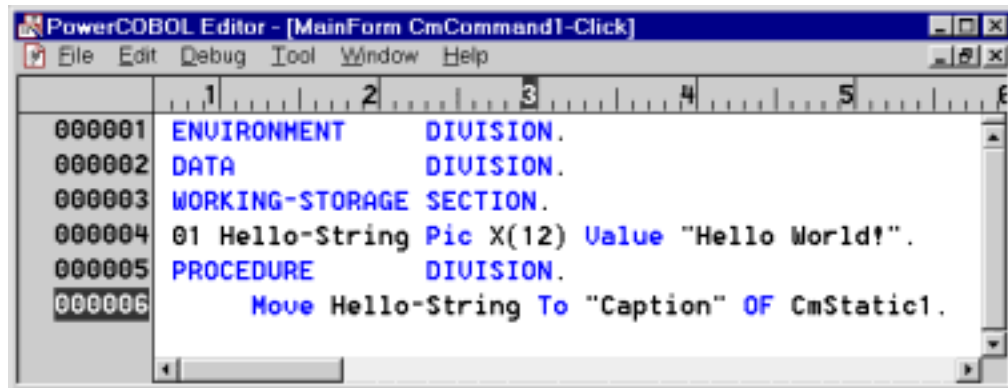


Figure 1.5. A sample event procedure

In this simple application, the string "Begin" initially shown in the form (window) in Figure 1.2 will be replaced with "Hello World!" when the user clicks on the push button labeled "Hello".

An event procedure has been defined to execute whenever the user clicks on the Hello push button. The Hello push button has been assigned the object name "CmCommand1" and the event is called "Click", so the title of the event procedure is "CmCommand1-Click", as shown in the title area of the Editor window in Figure 1.5.

If you examine the actual event procedure code, it is a very simple COBOL program. It contains a single line of execution code:

```
Move Hello-String To "Caption" of CmStatic1.
```

"CmStatic1" is the name assigned to the static text object on the form (window) that has an initial value of "Begin" (shown in Figure 1.2).

"Caption" is the property name of the actual text field managed by CmStatic1. Whenever the Caption property is assigned a new value, the text displayed in the top center of the form (window) will change to the new value.

To force the text in the form to change when the user clicks on the Hello push button, we have told PowerCOBOL that we are interested in the "Click" event for the Hello push button.

We have then created the single line of code shown above which moves a new value to the appropriate object property we are interested in changing ("Caption" of CmStatic1 - the text object on the form).

This single line of source code appears to be relatively standard COBOL, but in fact, a GUI object's property is being accessed directly using the scripting language, as "Caption" of CmStatic1 is not a standard COBOL identifier.

The event procedure code shown in Figure 1.5 above could actually be further simplified. There is not requirement that you use a data item to move data into the contents of a property such as "Caption" of CmStatic1.

Instead, you could move a literal string, and delete the Working-Storage data definition as shown below:

```
Move "Hello World!" To "Caption" of CmStatic1.
```

This approach to incorporating event-driven object programming with the COBOL language in a COBOL-like syntax greatly simplifies the application development process.

Users of PowerCOBOL are actually developing object-oriented GUI applications without being forced to learn the extraordinarily complex intricacies of object-oriented design and programming in a GUI environment.

The Object-Oriented Source Code Generated

When you save and build a PowerCOBOL project, using Object COBOL as the scripting language, an object-oriented COBOL program is generated, compiled and linked "under the covers" for each form in the application.

While you need never actually examine this object-oriented COBOL source code, you may if you like. When the application is built, a file with the name of each form and a .PRC extension will be created in one of the project's subdirectories depending on your current build mode.

Simply find the .PRC file and edit it to view it. For the sample Hello World application, whose project view is shown above in Figure 1.1, a single file named "Mainform.PRC" would be generated.

Connecting to Other Tools

As noted previously, PowerCOBOL takes advantage of the Microsoft Windows operating system by using Dynamic Data Exchange (DDE), a mechanism to exchange data automatically, and Object Linking and Embedding (OLE), to link or embed another application or pieces of another application directly into your application. Custom controls (OCX's), Microsoft Component (COM) objects, and even Microsoft ActiveX objects may be created and used in PowerCOBOL applications. You may also use the new *OLE class and *COM class to instantiate OLE/COM controls

and access a number of new and exciting technologies that are built upon the OLE/COM architecture.

Enabling New Fujitsu PowerCOBOL Controls.

Figure 1.2 above shows the PowerCOBOL Toolbox palette, which displays available controls. As previously noted, you may actually right click on this Toolbox palette and choose to add any number of additionally available controls to it. PowerCOBOL version 6.1 introduced two more advanced controls not included in the default Toolbox palette - the Fujitsu PowerCOBOL ADO Control, and the Fujitsu PowerCOBOL Microsoft Transaction Server Support Control.

If you desire to enable either of these controls to your Toolbox palette to make them available for PowerCOBOL applications development, simply right click the mouse on the Toolbox palette in the PowerCOBOL Form Editor, and select the Custom Controls option from the context menu that appears. Then scroll down and find these control and check the box next to them and they will be added to your Toolbox palette and thus enabled.

Developing Multimedia Applications

PowerCOBOL supports a wide range of data types, making it possible to build multimedia applications. PowerCOBOL supports sophisticated, multimedia types of data including:

- audio
- video
- animation

In the Microsoft Windows environment, the MCI string application programming interface (API) is used to control a wide array of multimedia devices. PowerCOBOL provides an MCI control object to allow direct access to this powerful API.

Developing Scalable Client/Server Applications

By embedding Standard Query Language (SQL) statements in COBOL code, you can use PowerCOBOL to access multiple, distributed databases. You can also write applications that support the client/server model.

PowerCOBOL allows you to code native SQL (EXEC SQL) statements directly in your application. Additionally, PowerCOBOL provides access to the ODBC application programming interface (API).

You may call stored procedures in SQL database systems via the ODBC interface as well.

The DB Access control allows you to connect to a wide range of local and distributed database systems using the widely supported and dynamic ODBC interface.

The PowerCOBOL ADO Data control supports Microsoft ActiveX Data Objects (ADO) database access methodology.

By supporting the open controls architectures of the Microsoft Windows environment such as ADO, OLE, DDE, and COM, (including Microsoft's Transaction Server), PowerCOBOL provides an optimal environment for creating scalable and widely distributed applications.

Make Use of the PowerCOBOL Sample Programs

PowerCOBOL ships with a significant number of sample programs that illustrate in detail how to use the individual controls, how to interact with a number of technologies, and even some complex samples.

It is strongly recommended that you examine these sample applications whenever you have a question regarding the usage of something included in the samples.

Appendix A contains a list and description of these sample applications. You will find these samples located in individual directories under the following default installation directory:

C:\Program Files\Fujitsu NetCOBOL for Windows\COBOL\Samples\PowerCOBOL\

These samples typically come as PowerCOBOL project files (.ppj), and will need to be loaded into the PowerCOBOL project Manager and built before executing them.

The subdirectories containing these samples are named meaningfully and will often be able to understand what the actual sample application refers to by looking at its subdirectory name.

Chapter 2. The Development Environment

This chapter describes the PowerCOBOL development environment including the Project Manager, the Form Editor, the Build facility, and the Debugger.

A more extensive explanation of using the Form Editor may be found in Chapter 4, "Creating and Editing an Application Window (Form)."

A more extensive explanation of using the Debugger may be found in Chapter 7, "Debugging the Program."

These facilities provide the following tools and features:

- Centralized management of all application components and properties
- Creation and editing of forms (windows) and event procedures
- Intelligent application build and rebuild
- Debugging applications
- Creation of custom application install (setup) programs
- Registering and de-registering applications with Windows
- Help menu

The PowerCOBOL Project Manager

The PowerCOBOL development environment is a graphical user interface with standard visual components. The development environment is centered on the PowerCOBOL Project Manager.

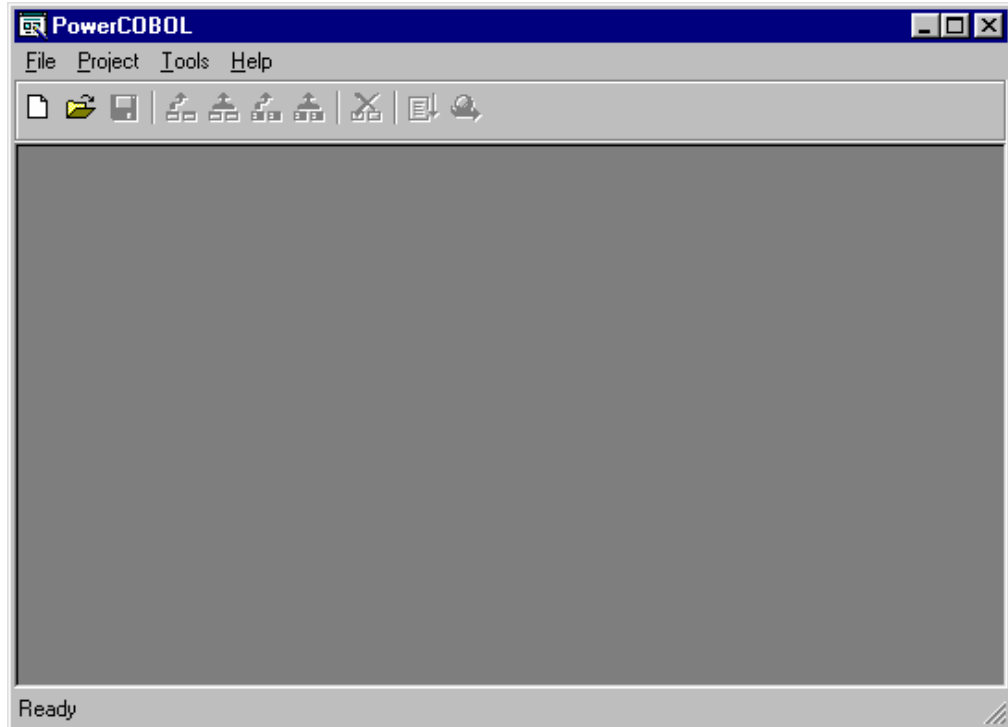


Figure 2.1. The PowerCOBOL Project Manager

From within the PowerCOBOL Project Manager, you may open an existing application or create a new application using one of the supplied application templates.

All details about an application are kept and managed in a special project file (.ppj file). Projects created with the previous release of PowerCOBOL (.prj files) are upwardly compatible with this version of PowerCOBOL and may be loaded into the Project Manager as well.

When you create a new application within the PowerCOBOL Project Manager (by selecting New from the File menu), you are presented with the New Project wizard dialog box as follows:

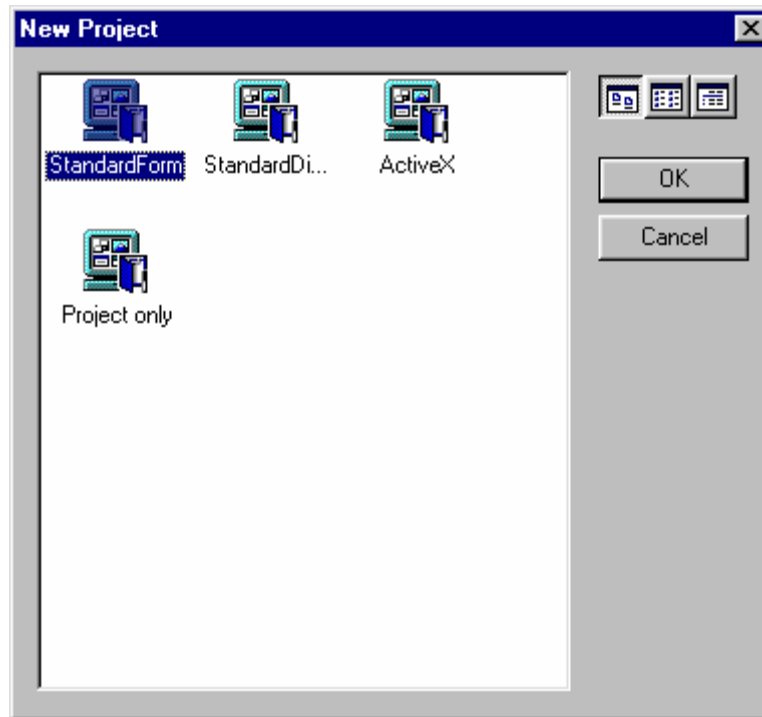


Figure 2.2. The New Project wizard dialog box

The New Project dialog box displays the project templates available to choose from. You may change the display format by clicking on any of the icons in the upper right portion of this dialog box.

In the example above, three project templates are available:

Standard Form - this creates a simple project containing a single module and a single form. The form created is entirely blank, and contains no controls.

Standard Dialog - this creates a simple project with a single module and a single form. Additionally, the form will have an "OK" and "Cancel" command button placed upon it. Both command buttons will additionally have a simple click event procedure created for it with a line of code deactivating it when clicked on.

ActiveX - this creates a project for an ActiveX control. It defines a module named "MyActiveX" and a form named "Control1". The scripting language becomes OO COBOL.

Project Only - this creates an empty project definition. The new project contains no objects such as modules or forms.

Once you create a new project or open an existing project, the Project Manager displays the application's components and their current properties as follows:

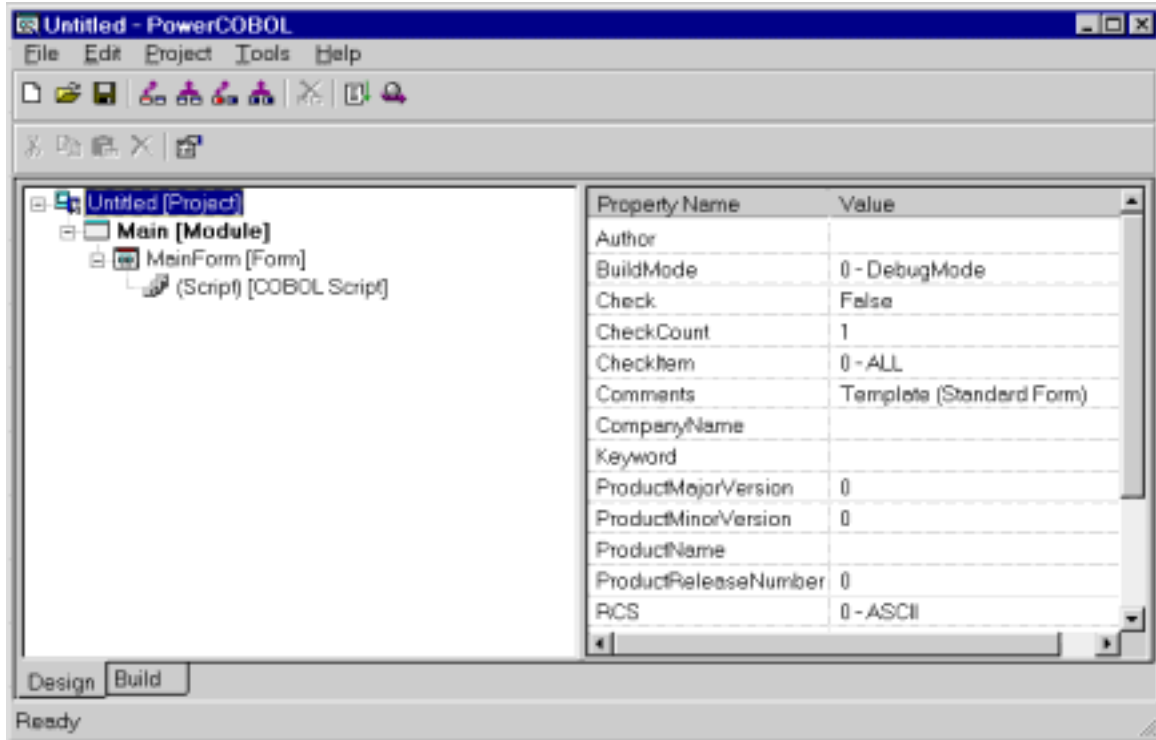


Figure 2.3. An application view in the Project Manager window of a Standard Form project created

NOTE: When a newly created or opened application is initially displayed in the Project Manager window, it is not expanded to show all levels of detail. You can actually set an option to tell the Project Manager what level of expansion you would like by default when a project is initially loaded by selecting the Design tab from the Tools menu's Options option. You can manually expand the view to show all levels of detail by right clicking the mouse on the project name and selecting Expand All from the pop-up menu. This shows all of the project's current application objects.

You control various development activities from the Project Manager window. If you highlight an application component in the left windowpane by selecting it, all of the current properties associated with it will be displayed in the right windowpane.

You can change these properties by clicking on them and typing new values or in other cases selecting values from a predefined list.

Also note that while many options are available from the various Project Manager pull down menus, you can access these options and many others by simply right clicking the mouse on a project component in the tree view and selecting one of the available options from the context menu that will pop up.

The Project Manager File Menu

File is the first menu on the menu bar of the PowerCOBOL Project Manager. It provides the following options:

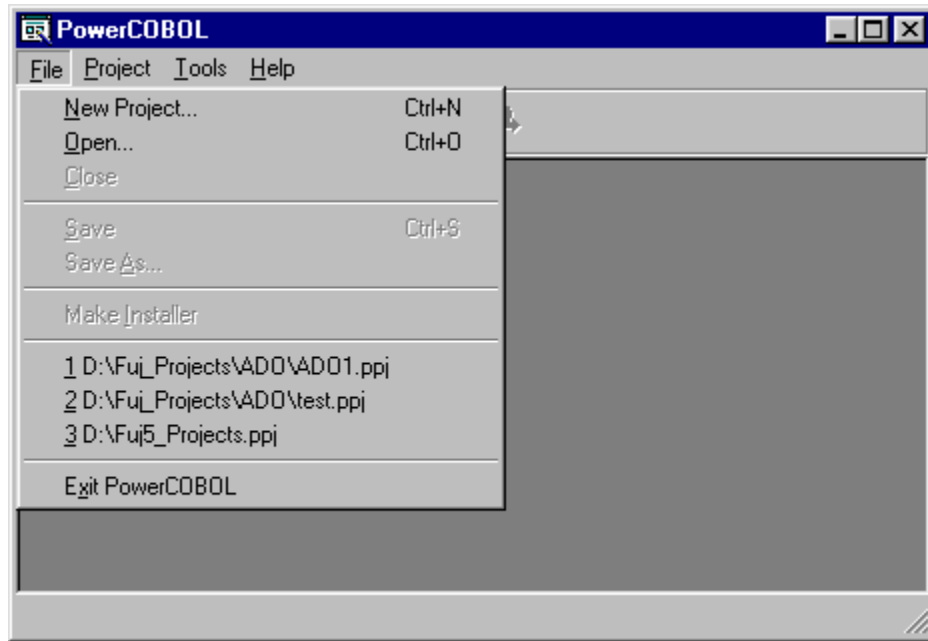


Figure 2.4. The PowerCOBOL Project Manager File menu

The File menu contains the following functions:

New Project

Displays the New Project wizard dialog box shown in Figure 2.2.

Open

Opens an existing application (project file). The default project file type extension for this is .ppj for version 4 projects. If you are opening a file created with the previous version 3 of PowerCOBOL, make sure you select the "Files of type:" dropdown menu and click on the "PowerCOBOL V3 or earlier project (*.prj)" file extension type.

Close

Closes the currently opened project.

Save

Saves the application (project) you are currently working on.

Save as

Saves the current project (application) you are currently working on under a different name or in a new path (in a different directory or on a different drive).

Make Installer

Creates a complete setup program to install your application on another machine, including a Windows .inf file and an un-install file to remove your application. The file created (Setup.EXE) will be placed in the same directory as your main application module .exe or .dll. This is typically in the Debug or

Release subdirectory depending on your current build mode specified in the project's properties.

Exit PowerCOBOL

Ends the current PowerCOBOL session.

NOTE: The File menu also maintains the name of the last four applications (project files) that you had open and allows you to select one to open directly from here.

The Project Manager Edit Menu

The Edit menu appears only if you have a project currently open

The Edit menu appears only if you have a project currently open and gives you access to functions in the PowerCOBOL Project Manager that may assist you in shaping the design of your application. Accessibility to functions in the Edit menu varies depending on what type of application component you have currently selected in the left windowpane. Note that the Edit menu only appears when you have a project loaded in PowerCOBOL.

The Edit menu appears as follows:

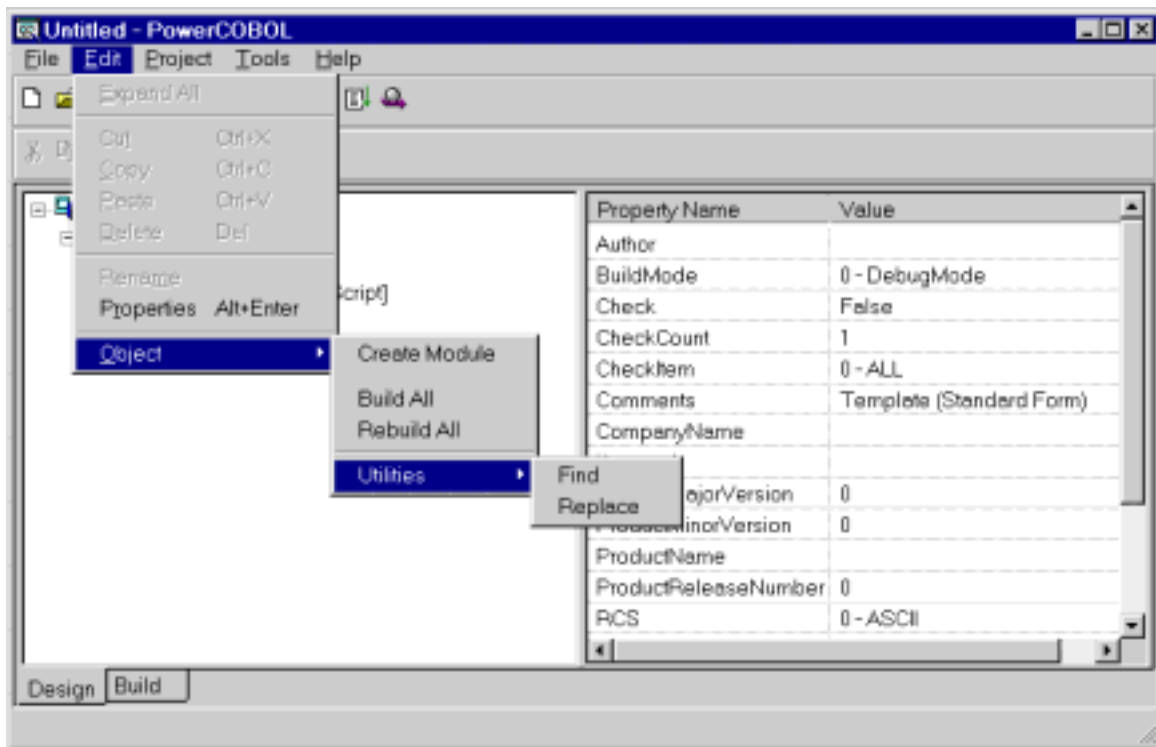


Figure 2.5. The PowerCOBOL Project Manager Edit menu

The Edit menu contains the following functions:

Expand All

Expands the project hierarchy to show all levels of the tree.

Cut

Removes the selected application component and stores it on the clipboard. At the next Cut or Copy command, the selection is erased from the clipboard.

Copy

Copies the selected application component to the clipboard without removing the selection from the current project definition.

Paste

Places the clipboard contents into the current project under the appropriate managing component in the hierarchical project tree. For example, if you have copied a form (window) and wish to insert it into the current project to create a separate identical form, you would insert this new form directly under the application module that will control the form. **Note:** The Paste command usually follows the Copy or Cut command.

Delete

Removes the selected application component from the current project definition.

Rename

Renames the selected application component in the current project definition.

Properties

Displays the Properties dialog box for the currently selected application component. This dialog box allows you to examine and modify properties similar to the Properties list in the right windowpane. The Properties dialog box differs from the Properties list in that it organizes related properties, whereas the Properties list presents all properties in alphabetical order. The Properties dialog box appears as follows:

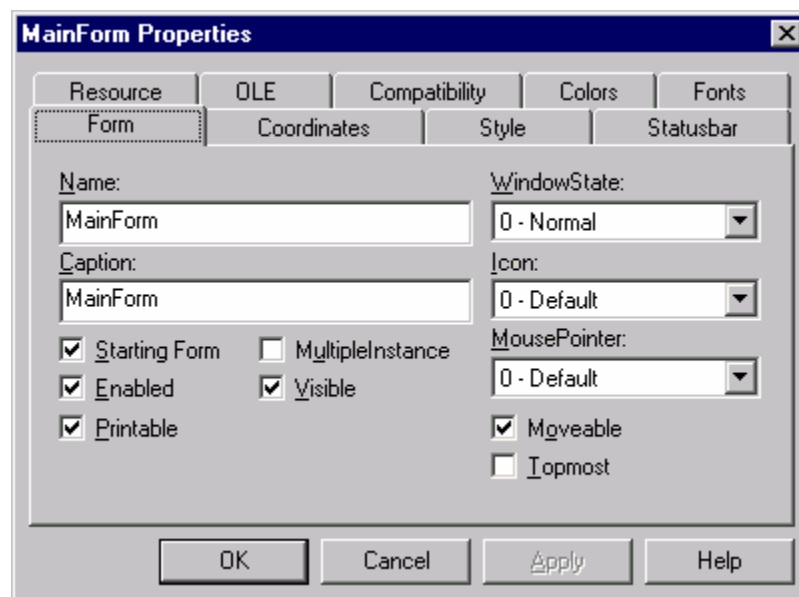


Figure 2.6. The Form Properties dialog box

Object Options in the Edit Menu

A number of sub options are available from the Object selection on the Edit pull down menu. You may also right click on any object in the Project Manager Hierarchy and a context menu will appear containing options. The options found in the Edit->Object menu change dynamically depending upon the type of application object currently selected (highlighted) in the Project Manager hierarchy as noted below:

Object Menu Options (For a Project)

Figure 2.5 above shows the Edit->Object menu expanded for a Project (e.g. when the actual project name is currently selected in the Project Manager hierarchy). The options available include:

Create Module

Allows you to add a new executable module definition in your project. PowerCOBOL projects can create one or more executable modules.

Build All

Builds the project, compiling and linking only components that were changed.

Rebuild All

Rebuilds (compiles and links) all application components, regardless as to whether they were modified.

Utilities

Brings up the Find and Replace utility context menu. These utilities allow you to search through an entire project looking for search strings you may specify, and to optionally replace the search string with a new string.

Expand All

Expands the entire project hierarchy.

Properties

Brings up the Properties dialog box for the item currently selected in the project hierarchy.

Object Menu Options (For an executable Module)

When you select (highlight) an executable Module in the Project Manager hierarchy and select the Edit->Object menu, the following options are available to you:

Create Form

Creates a new form in the module.

Insert File

Allows you to insert an external file into the current PowerCOBOL project. External files supported are COBOL programs, Object files (.OBJ), Library files (.LIB), Bitmap files (.BMP), Icon files (.ICO), Cursor Files (.CUR), Image Lists (.BMP), Animate Cursor files (.ANI). If your executable module required the static linking of an external .OBJ or .LIB file that had been created outside of PowerCOBOL, for example, this is the option you would use to add it into your project.

Build

Invokes the Build facility.

Rebuild

Invokes the Rebuild facility.

Debug

Invokes the PowerCOBOL Debugger on the currently select module

Execute

Executes the currently selected module.

Register

Registers the currently selected module in the Windows Registry (invokes the Windows Regsvr32.exe utility).

Unregister

Unregisters the currently selected module from the Windows Registry (invokes the Windows Regsvr32.exe utility).

Utilities

Brings up the utilities context menu containing the Find and Replace utilities.

Expand All

Expands the Project Manager Hierarchy completely under the currently selected module.

Cut, Copy, Paste Delete

Standard Windows cut, copy paste and delete operations.

Rename

Allows you to rename the currently selected module.

Properties

Brings up the module's Properties dialog box.

Object Menu Options (For a Form)

When you select (highlight) a Form in the Project Manager hierarchy and select the Edit->Object menu, the following options are available to you:

Open

Opens the currently selected Form in the PowerCOBOL Form Editor, allowing you to modify it.

Compile

Compiles the currently selected Form. Note that "under the covers", PowerCOBOL creates a separate program for each Form defined in a module. A PowerCOBOL executable Module is made up of one or more Forms compiled and linked together.

Edit ENVIRONMENT DIVISION

Allows you to modify the SPECIAL-NAMES, REPOSITORY, and FILE-CONTROL sections of the ENVIRONMENT DIVISION of the currently selected form. The FILE-CONTROL section is where you would place SELECT....ASSIGN TO statements to define an external data file to be accessed from a Form, for example.

Edit DATA DIVISION

Allows you to modify the BASED-STORAGE, FILE, WORKING-STORAGE, and CONSTANT sections of a Form. The FILE section, for example, is where you would place FD statements and associated record definitions for external data file definitions to be accessed from the Form. The WORKING-STORAGE section is where you would define COBOL data items for the Form. Note that if you want to share these data definitions (both FD's and WORKING-STORAGE section items) with event procedures in your Form, you must specify the IS GLOBAL optional phrase on each such definition. To share these definitions across separate executable modules, you must specify the optional IS EXTERNAL phrase on each such definition.

Edit PROCEDURE DIVISION

Allows you to create COBOL programs (scriptlets) that are not tied to any particular event and may be called from other event procedures or scriptlets. If you wanted to create a program to format dates for example and wanted to call it from several different places in your PowerCOBOL application, this is where you would define it. You create a new scriptlet by selecting the New option that appears. The PROCEDURE option allows you to specify declaratives and SQL cursor definitions so that they get executed first when the form is initialized.

Edit Event Procedures

Shows a list of all of the event procedures that are available for the Form, and allows you to edit them. Event procedures are individual COBOL programs (also called "scriptlets") that you write to handle specific events, such as when a user clicks the mouse on a specific command button. The list presented here, however, are event procedures associated with the form itself - not the controls that you may have placed on the form. For example, you might want to use the "Opened" to write initialization code, as it gets triggered when the form is opened. If you see an asterisk character to the left of one of the event procedures in the context menu that appears, it designates that an event procedure currently exists for that event.

Preview

Previews the currently selected Form. This brings the form up on the desktop as it will look at execution time.

Tab Order

Allows you to define the tab order of controls on the form.

Insert Custom Property

Allows you to create a custom property for the form and to define its data type.

Insert Custom Method

Allows you to create a custom method and define its parameter usage.

Insert Custom Event

Allows you to create a custom event.

Menu Editor

Allows you to create or modify a menu for the form.

Print

Allows you to print information regarding the form.

Utilities

Brings up a child menu from which you can access the Find and Replace utilities.

Expand All

Expands the entire project hierarchy.

Cut

Standard Windows Cut function.

Copy

Standard Windows Copy function.

Paste

Standard Windows Paste function.

Delete

Standard Windows Delete function.

Rename

Allows you to rename the form.

Properties

Brings up the form's Properties dialog box.

Object Menu Options (For Script)

When you select (highlight) the Script(COBOL Script) entry in the Project Manager hierarchy and select the Edit->Object menu, the following options are available to you:

Print Procedures

Allows you to print the source code of existing event procedures.

Utilities

Brings up the utilities context menu allowing access to the Find and Replace utilities.

Compile

Compiles the event procedures.

Edit ENVIRONMENT DIVISION

Allows access to the form's ENVIRONMENT DIVISION sections.

Edit DATA DIVISION

Allows access to the form's DATA DIVISION sections.

Edit PROCEDURE DIVISION

Allows access to the form's PROCEDURE DIVISION.

Properties

Brings up the Properties dialog box for the currently selected item in the project hierarchy.

The Project Manager Project Menu

The Project Menu provides access to project oriented tools and appears as follows:

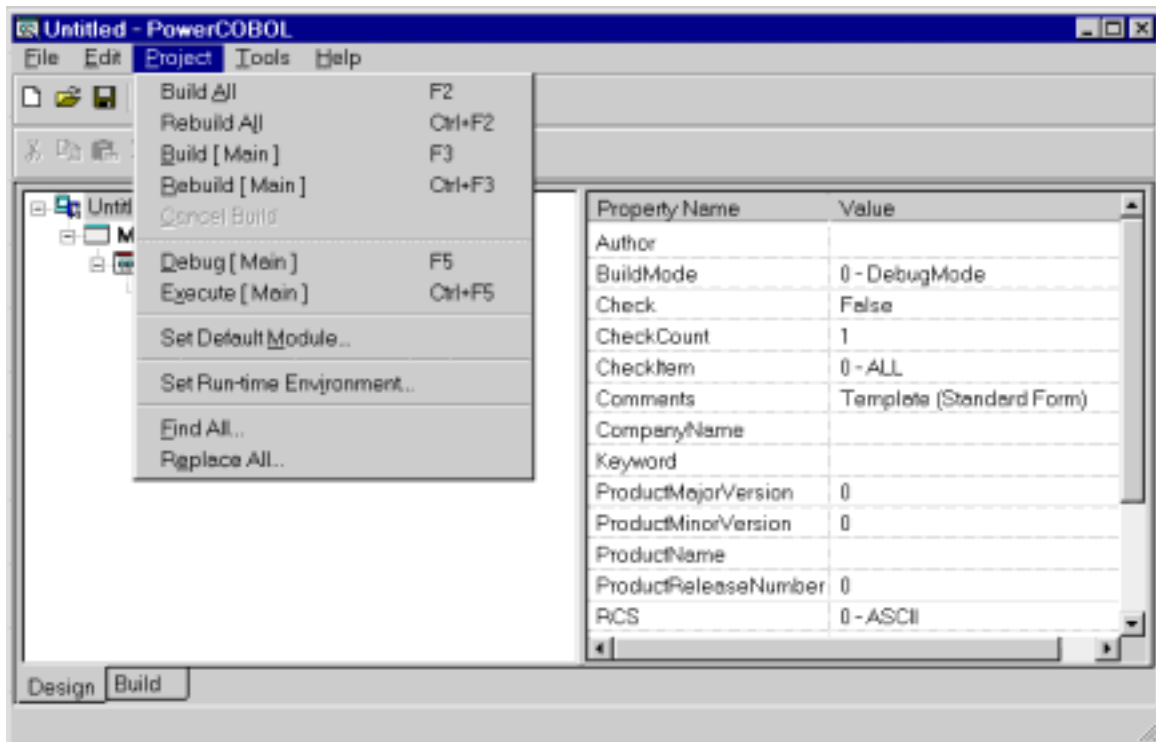


Figure 2.7. The Project menu

The options available from the Project menu include:

Build All

Builds (compiles and links) all modules in the project. It first examines the date and time stamp of each file to determine if it has changed since the last build process and only re-compiles and re-links programs that have changed.

Rebuild All

Rebuilds all modules in the project, re-compiling and re-linking them regardless as to whether they have changed or not since the last build operation.

Build [module name]

Builds only the default module, re-compiling only those programs that have changed and re-linking the module only if a program or resource has changed.

Rebuild [module name]

Rebuilds the default module, re-compiling all programs contained therein and re-linking it regardless as to whether anything has changed since the last build operation.

Cancel Build

Stops a current build or rebuild in process.

Debug [module name]

Starts the PowerCOBOL debugger on the default module.

Execute [module name]

Executes the default module.

Set Default Module

Allows you to select which module will be executed first when you execute a project containing multiple modules.

Set Run-time Environment

Brings up the Runtime Environment setup tool.

Find All

Brings up a dialog window that allows you to search a project for occurrences of strings in your COBOL event procedures (scriptlets).

Replace All

Brings up the dialog window shown in Figure 2.8 below that allows you to search an entire project for and to replace strings in your COBOL event procedures:

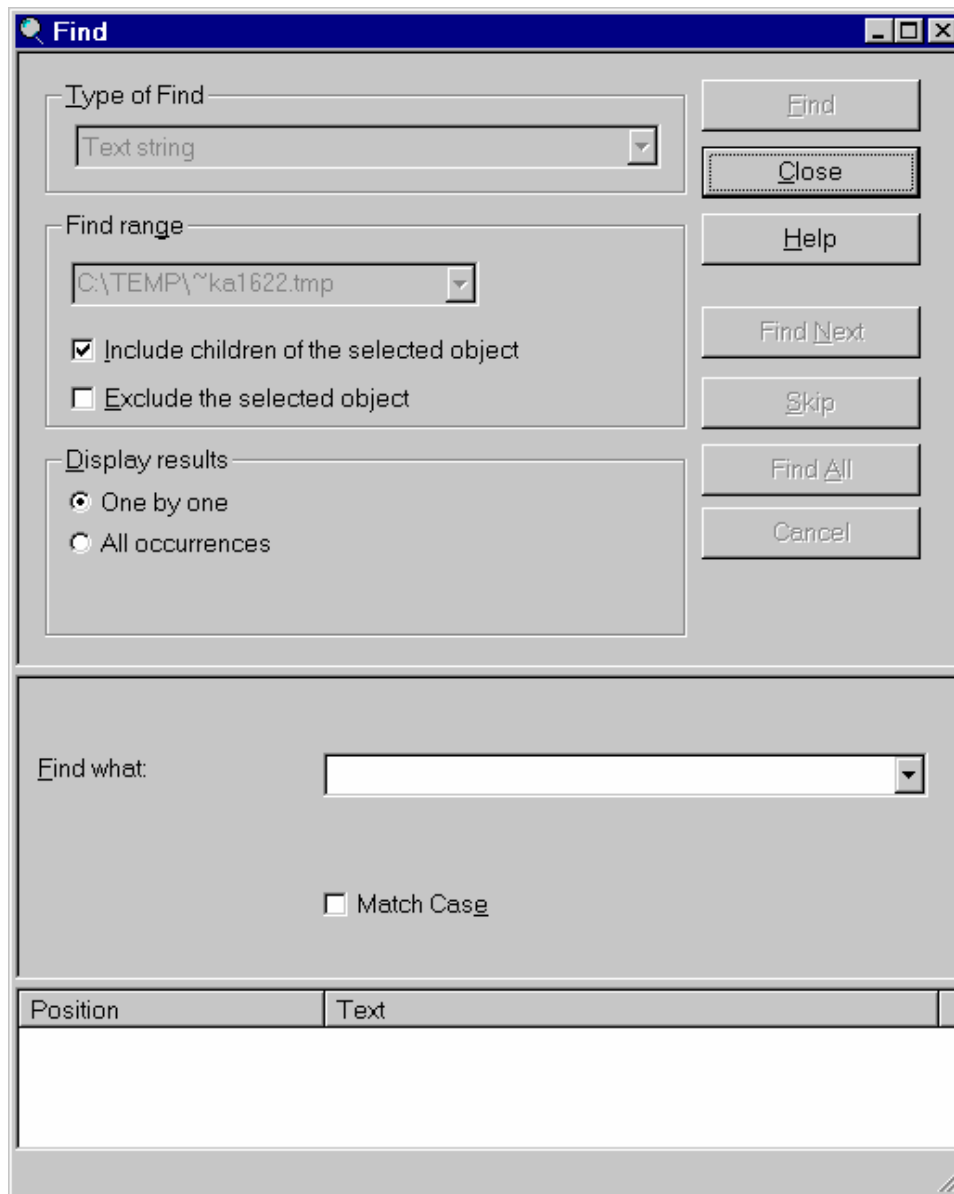


Figure 2.8. The project Find All dialog window.

The Project Manager Tools Menu

The Tools Menu provides up an Option dialog box that allows you to customize the Project Manager environment as follows:

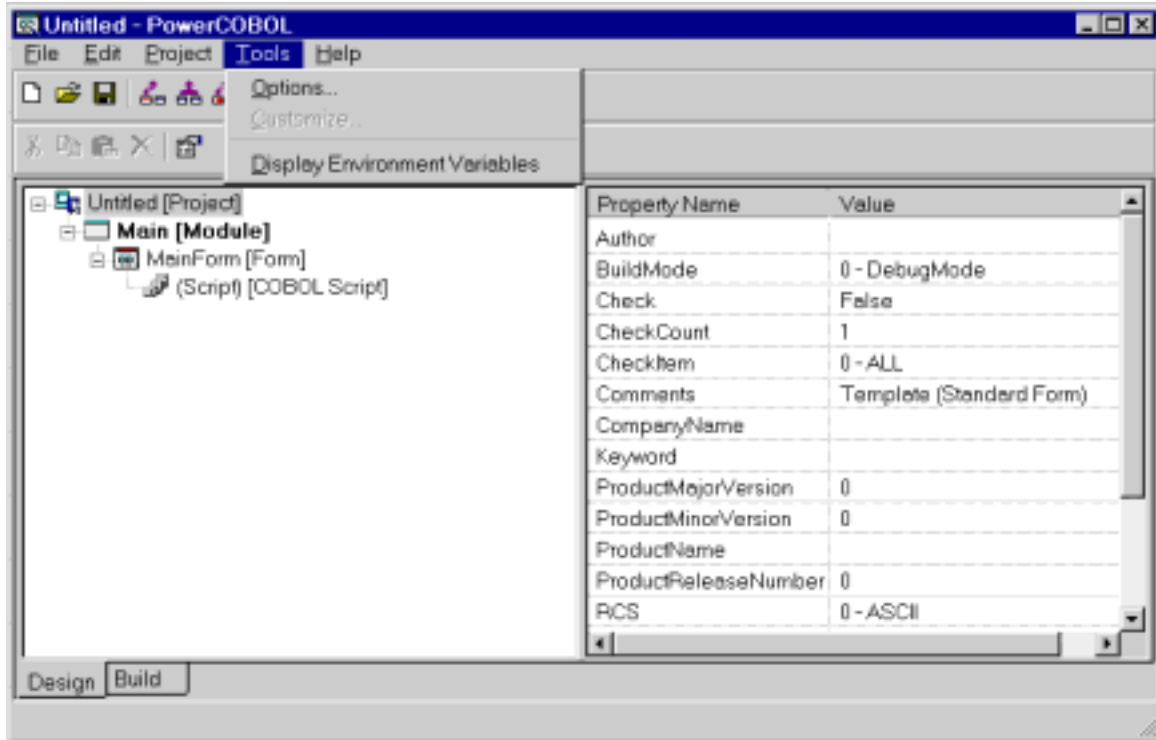


Figure 2.9. The Tools menu

The options available in the Tools menu are:

Options

This brings up the Option Properties dialog window where you can set a number of PowerCOBOL options such as automatic backup, build options, design options, form options, editor options, and debug options.

Customize

This option allows you to add in your own default editor for PowerCOBOL instead of using the PowerCOBOL editor. It is only available when no project has been loaded.

Display Environment Variables

Displays the current value of certain Fujitsu environment settings that affect the project building.

Project Manager Options

The options that you can configure from the PowerCOBOL Project Manager have been extended. The following sections describe the new options.

You display the project options by selecting Options from the Tool menu. Project Manager now displays a dialog with six tabs: Project, Build, Design, Form, Editor, Debug. The Project tab contains the project options as in earlier releases; the other tabs contain new Project Manager options.

Project Options

The figure below shows the Project Options tab:

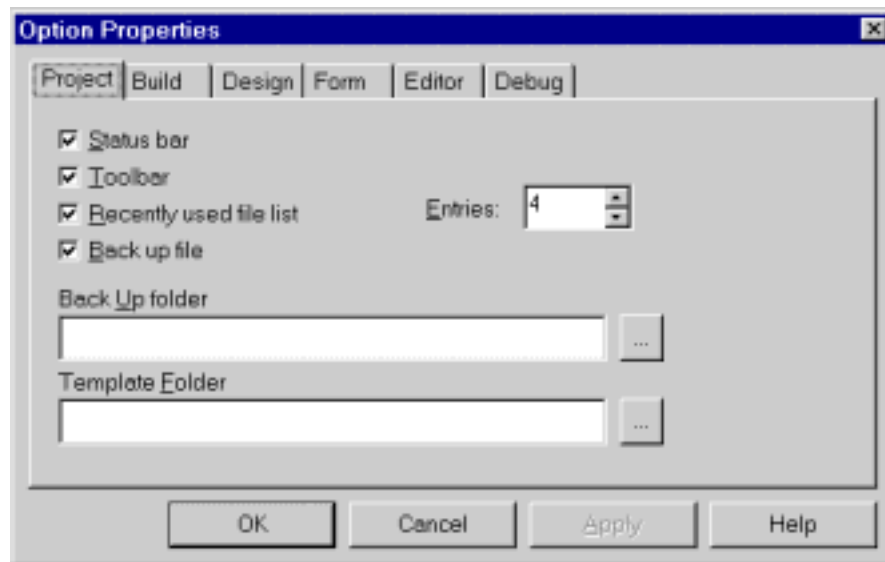


Figure 2.10. Project Option Properties

Where:

Status Bar

Turns on/off the status bar displayed at the bottom of the Project Manager Window

Toolbar

Turns on/off the Project Manager toolbar

Recently used file list

Turns on/off the display of the most recently access .ppj files in the File pull down menu

Back up file

Activates/deactivates the automatic project backup (.ppj~ file will represent the previous version of the project)

Back Up Folder

Specifies the path where you want the back up version of the project placed. If left blank, it will be placed in the same folder that the project was loaded from.

Template Folder

Specifies the path of the folder containing the template projects. If left blank, it points to the default template folder (C:\Program Files\Fujitsu NetCOBOL for Windows\COBOL\STDTMPLT\)

Build Options

The figure below shows the Build tab:

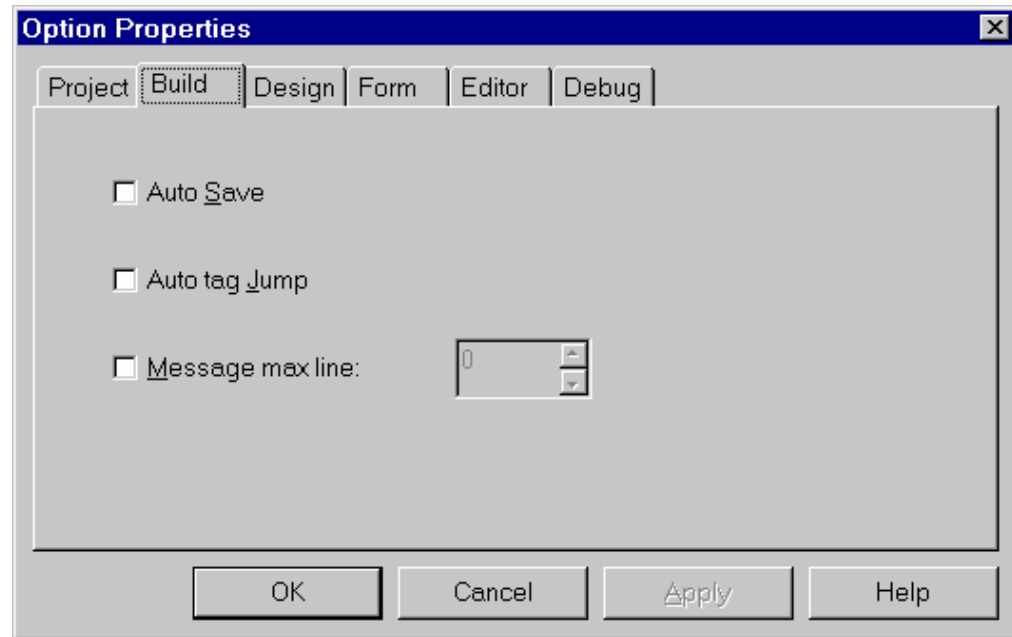


Figure 2.11. Build Option Properties

Where:

Auto Save:

Specifies whether to save projects automatically before you build them.

If you check this option, the build-time message asking you to confirm whether the project should be saved is not displayed.

Auto tag Jump:

Specifies whether PowerCOBOL should automatically display the first line of code containing an error if a build error was caused by a coding error.

If you check this option PowerCOBOL opens the editor at the first line containing an error when the build finishes.

Message max line:

Specifies whether to stop the build when a specified number of error messages have been displayed.

If you check this option the build is interrupted when the number of error messages reaches the number specified.

Design Options

The figure below shows the Design tab:

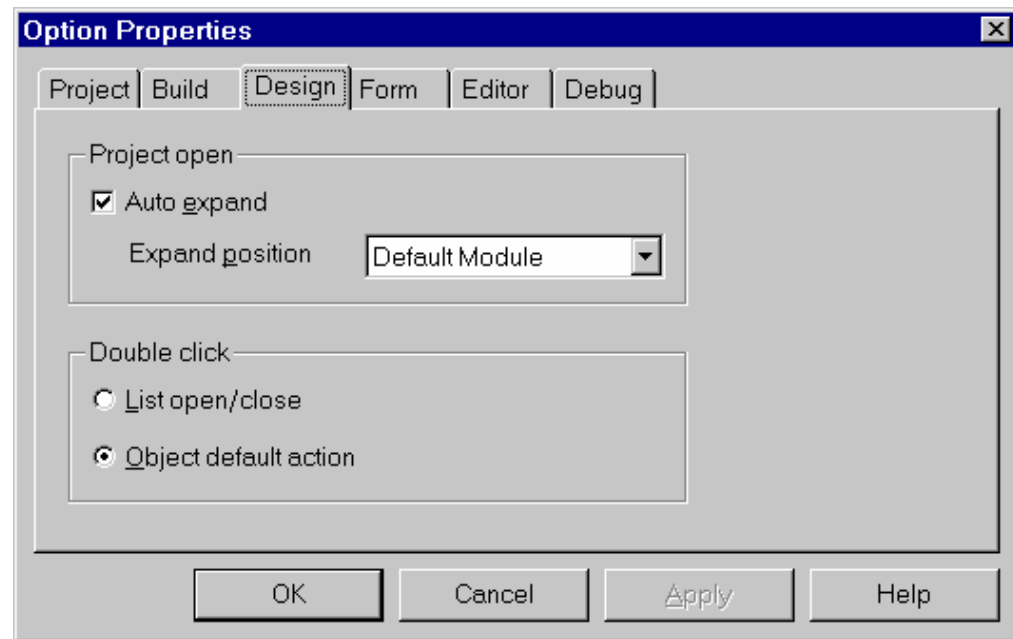


Figure 2.12. Design Option Properties

Where:

Auto expand

Specifies whether or not the Project Manager should automatically expand the tree hierarchy displayed in the left windowpane when a project is initially opened. If this box is checked (on), then the selection in the Expand position drop down list will be used to determine what level in the hierarchy to expand to.

Double click

Specifies the behavior when you double-click on an item in the project tree pane of the Design window.

If you select "List open/close", double-clicking opens or closes branches on the project tree.

If you select "Object default action", PowerCOBOL behaves as follows:

Object type	Double-click default behavior
Form	Displays the form edit window.
Menu	Displays the menu edit window.
Scriptlet	Displays the procedure edit window.
COBOL file	Displays the procedure edit window.
Others	Displays the property setting window.

Form Options

The figure below shows the Form tab:

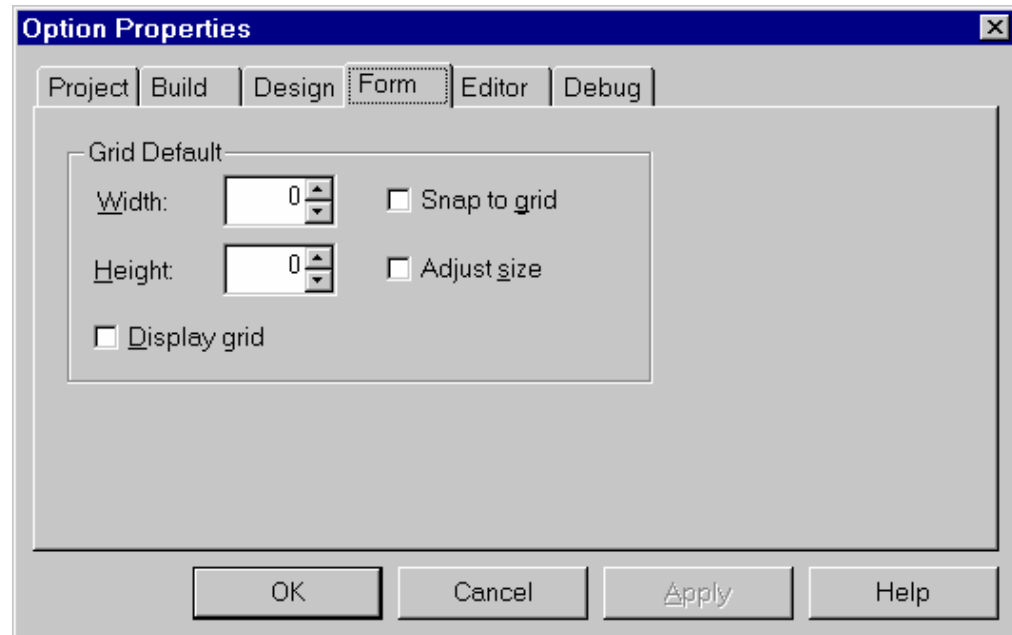


Figure 2.13. Form Option Properties

Where:

Grid Default:

Specifies the default values to be used for the grid when you create a new form.

Width / Height:

Specify the width and height of the grid in points, from 0 to 99.

Display grid:

Specifies whether to display the grid on the form.

Snap to grid:

When you select "Snap to grid", the Form Editor ensures that the upper left hand corner of a control being positioned matches a point on the grid.

Adjust size:

When you select "Adjust size", the Form Editor ensures that, when you adjust the width or height of a control, it is an integer multiple of the grid width or height.

Editor Options

The figure below shows the Edit tab:

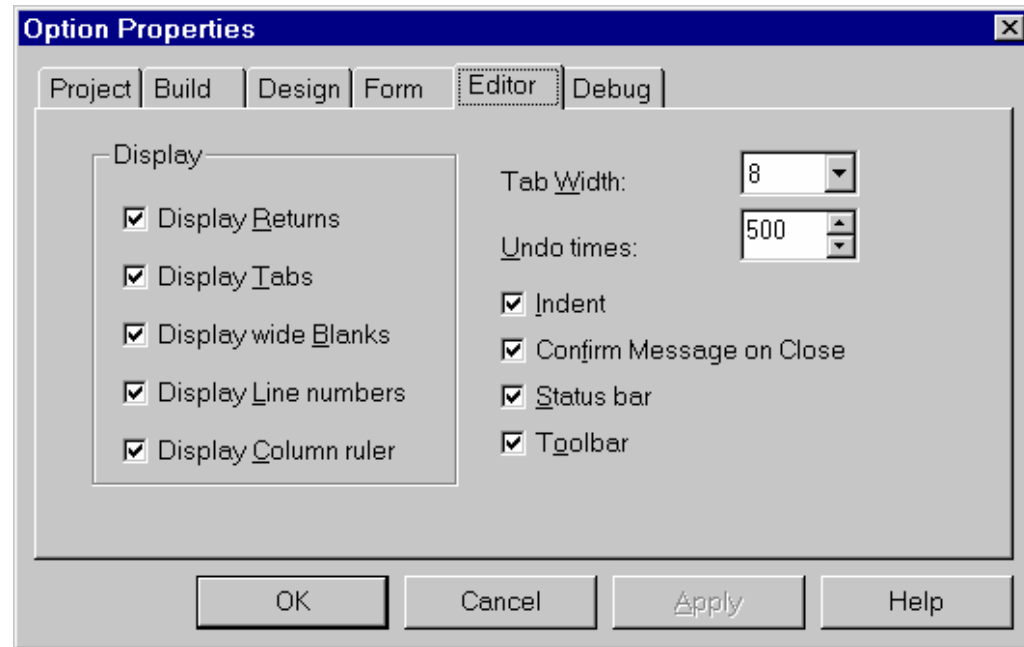


Figure 2.14. Editor Option Properties

Where:

Display Returns, Display Tabs and Display wide Blanks:

Specifies whether to display tab, line feed and em space characters in the PowerCOBOL Editor window.

Display Line numbers and Display Column ruler:

Specifies whether to display line numbers and a column ruler in the PowerCOBOL Editor window.

Tab Width:

Specifies whether tabs move to the column that is the next multiple of 4 or 8.

Tabs cannot be used for moving to the head of Area A or B.

Undo times:

Specifies the number of operations that can be undone using the Undo function. A very high number may affect the performance of the editor.

Indent:

Specifies whether to start the next line under the first character of the previous line when the ENTER key is pressed. When Indent is not selected the cursor returns to the left-most column when ENTER is pressed.

Confirm Message on Close:

Specifies whether to display a "Save Changes?" message when a PowerCOBOL editor window is closed and the window contained unsaved changes.

If you do not check "Confirm Message on Close", the editor saves the procedure and closes the window without displaying a confirming message.

Status bar

Specifies whether or not to display the status bar at the bottom of the Editor window.

Toolbar

Specifies whether or not to display the Editor's toolbar.

Debugging Options

The figure below shows the Debug tab:

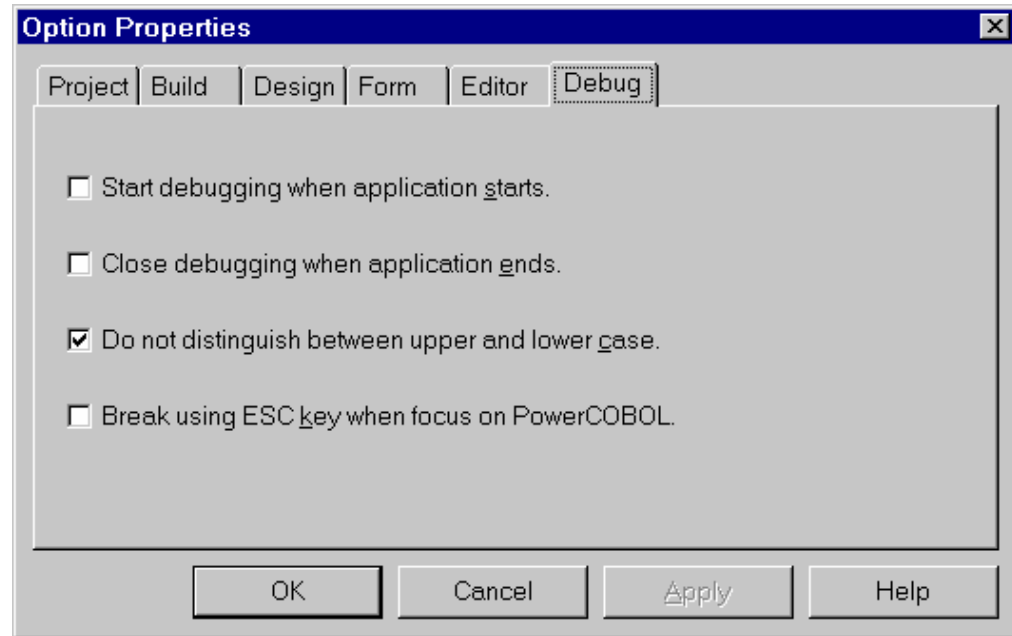


Figure 2.15. Debug Option Properties

Where:

Start debugging when application starts:

Specifies whether the debugger starts the application automatically. When checked, you can only interact with the debugger once the application has started. When not checked, you can interact with the debugger before the application starts.

Close debugging when application ends:

Specifies whether to close debugging when the application ends.

Do not distinguish between upper and lower case:

Specifies whether to treat upper case and lower case the same when interpreting data and procedure names.

Break using ESC key when focus on PowerCOBOL:

Enables the ESCape key to break execution when debugging.

The Project Manager Help Menu

The Help menu provides a list of PowerCOBOL topics to assist you in using PowerCOBOL. It also provides an About option that displays the version and build numbers of your current PowerCOBOL installation.

The Help facility is extremely valuable for searching for available methods and properties for controls to determine their usage, possible data values and data formats.

It is strongly recommended that you make frequent use of the help system, as it is the quickest way to obtain information while designing projects. It contains definitions and examples of all methods, properties and events, as well as code samples.

The PowerCOBOL Form Editor

The PowerCOBOL development environment provides a powerful form editor for designing graphical user interfaces (GUI's).

The Form Editor (Design facility) allows you to create forms (windows) with a rich set of graphical controls, to access other sharable controls and objects in your Windows system, and to manage the development of event procedures.

When you create a new project (application) using the PowerCOBOL New Project wizard, you will typically have a default form (window) created automatically. To invoke the Form Editor on an existing form, right-click the mouse on the name of the form in the left windowpane of the Project Manager and select Open from the pop-up menu that appears. If you wish to create an additional form for an application, right-click the mouse on the name of the module that will manage the form. Then select Create Form from the pop-up menu. A new empty form will be created and the Form Editor will automatically appear with this new form.

When you open a form for editing, the Form Editor will appear as follows:

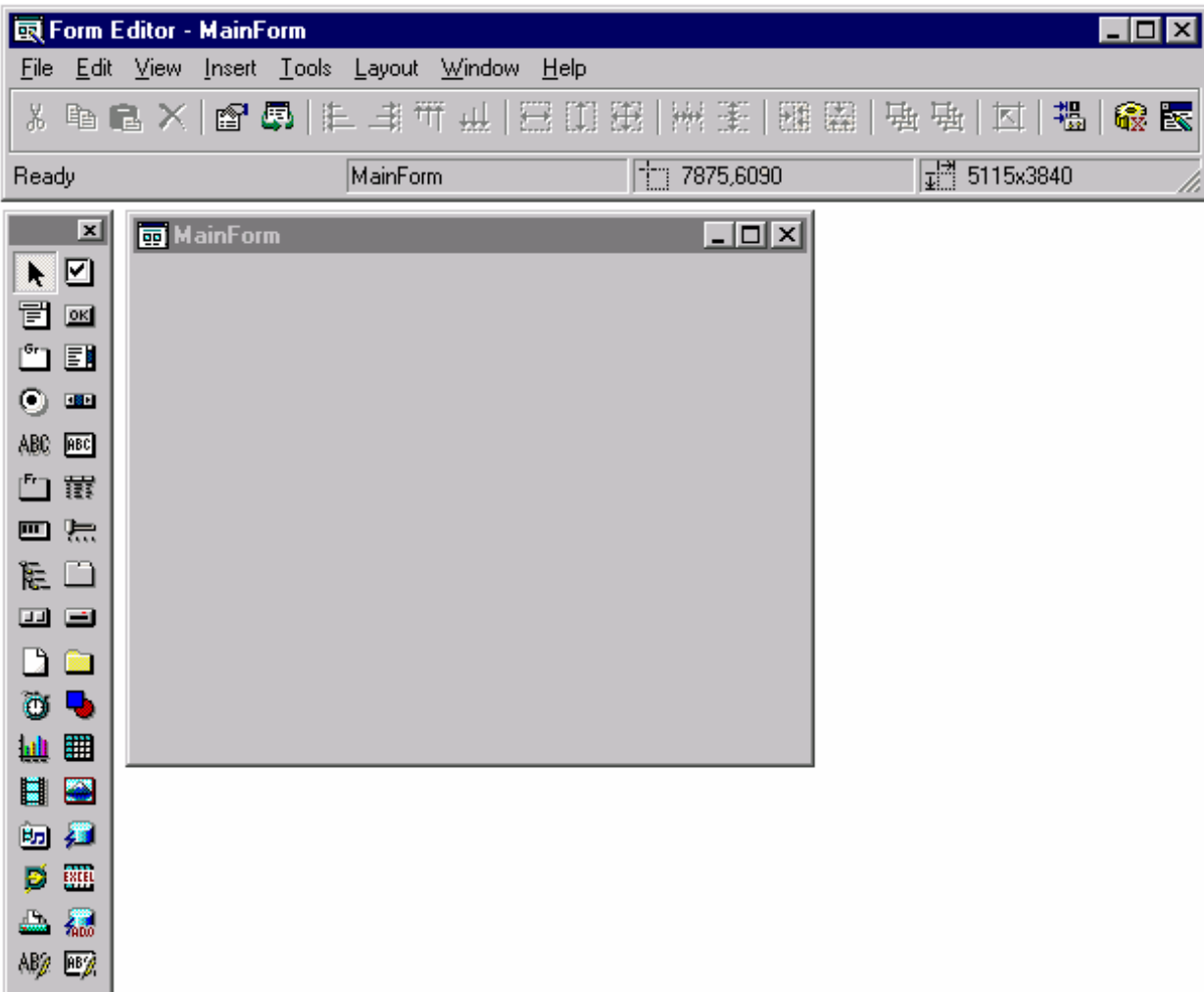


Figure 2.16. The PowerCOBOL Form Editor

The Form Editor consists of three separate windows.

The top window is the control bar, which contains a number of pulldown menus, associated action icons and a status bar area beneath the action icons.

The blank window on the right is the actual form that you are currently editing (this process is also known as form painting).

The window to the left is the Toolbox palette. It shows all of the controls available directly within PowerCOBOL.

You select a control by left clicking on its icon, dragging the mouse over to the position on the form where you wish to place the control, and left clicking a second time to drop the control on the form. This is known as *drag and drop* programming.

Controls directly available from the Toolbox palette are explained in Chapter 1, "Introduction."

Chapter 4, "Creating and Editing an Application Window (Form)" explains in detail how to use the Form Editor.

The PowerCOBOL Build Facility

PowerCOBOL provides developers with an intelligent application build facility.

Building an entire application from the Project Manager is as easy as right clicking the mouse on the project name and selecting Build All from the pop-up menu.

If you instead want to build only a portion of the application (for example, a specific application module), you right click on the module you want to build and select Build.

The Build facility examines all components associated with the current build request to determine the appropriate action(s), if any, to take on each of them.

For example, if an application program has not been changed since the last build took place, there is no need to re-compile it, and the Build facility will skip the compile step and move instead directly to the link step.

You can override this process by instead selecting Rebuild All or Rebuild from the pop-up menu, which forces a re-compilation of all components associated with the current build request, regardless of their current state.

If the Build All tool is invoked, the currently selected module is compiled and linked into an appropriate executable.

If the Rebuild All tool is invoked, the entire application is compiled and linked into an appropriate executable.

If you previously selected debug mode for building the application, the application is ready for use with the Debugger. You may optionally execute any built application.

If the build process encounters an error, the Build window tab will appear within the Project Manager window. The Build window is actually a tab available at any time within the Project Manager.

Each error encountered during the build process will be listed and highlighted in red as illustrated in the following figure:

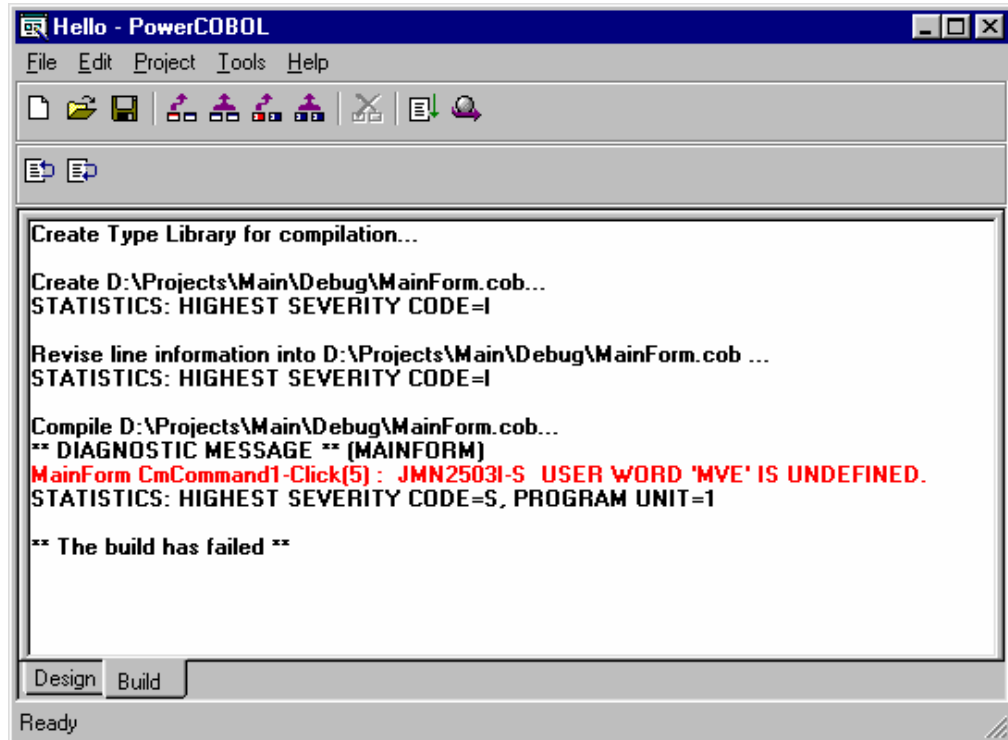


Figure 2.17. The Build window illustrating that an error has been encountered

The Build facility now becomes a very convenient tool for correcting errors throughout the application. You may move the mouse to any error (any line highlighted in red) and double click the left mouse button on it.

This will display the actual application source code in an edit session, and position you automatically on the line of code associated with the current error. If you were to do this for the error message shown in Figure 2.12, you would be presented with the following:

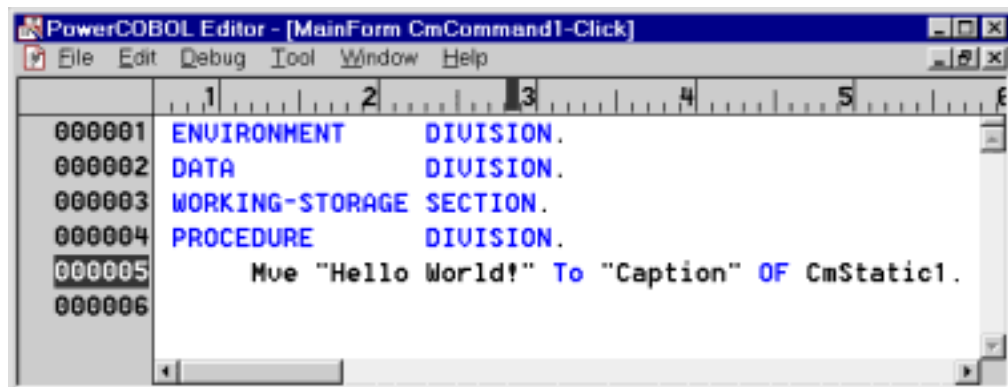


Figure 2.18. An edit session on the source code causing a build error

In this case, you would simply change the erroneous spelling of the Move statement in line 5, save the change and close the edit session. You would then be ready to rebuild the application.

The Build facility offers a number of options for correcting errors. Under the Edit menu, you will find the Next Error and Previous Error options. This allows you to move quickly between multiple errors encountered during the build in order to correct them.

You may optionally use the Jump to Previous Error and Jump to Next Error icons located on the right side of the tool bar to perform the same correction tasks.

Once you have jumped to an error message and opened and closed an edit session on the related source code, the red error highlight changes to blue to show that you've accessed the erroneous code.

Under the Build menu, there is an option called Auto Jump that you may check on or off. When this option is enabled, you will automatically be taken to the erroneous section of source code that caused the first error encountered during the build.

Once you have corrected the errors, you have a number of options to re-start the build process. For example, you may select an individual module to build or rebuild to ensure you've corrected all of its local problems without invoking the Build All or Rebuild All process for the entire application.

You may invoke Build, Rebuild, Build All or Rebuild All from the Build menu, or by selecting any of the associated icons on the tool bar.

You may additionally move back and forth from the Build facility to the Design facility freely at any time by simply left clicking the mouse on the appropriate tab near the bottom left corner of the Project Manager window.

The PowerCOBOL Debugger

PowerCOBOL provides a powerful and highly integrated debugger to assist in developing applications.

You must build an application in debug mode in order to use the Debugger on it. This is the default mode for a new application.

To select debug mode for building the application, you select and change the project's Build Mode property to 0-DebugMode. You can do this by highlighting the project name in the Project Manager and selecting the BuildMode Property in the right windowpane.

You may also select the BuildMode property by right clicking the mouse on the project name and selecting Properties from the pop-up menu. You will then be presented with a Properties dialog box from which you can select the Build tab as follows:

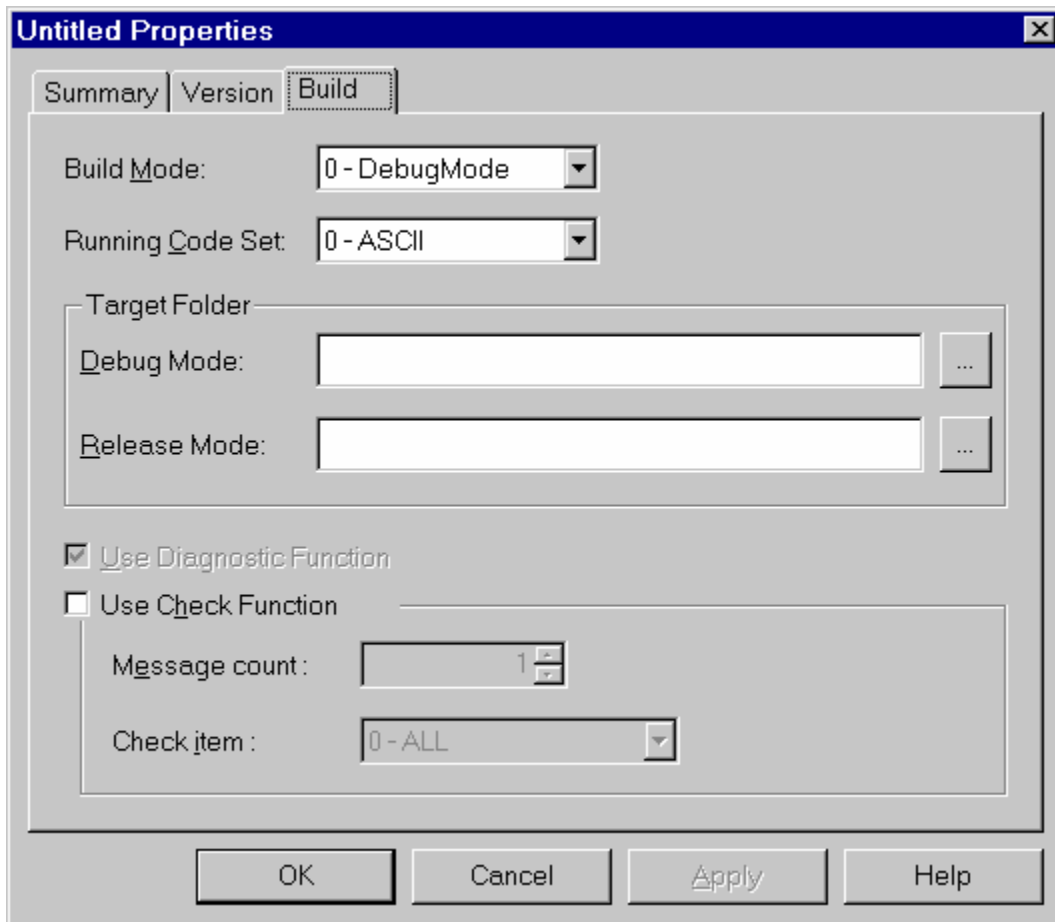


Figure 2.19. The Build tab in the application's Properties dialog box

You may then select the Build Mode dropdown list and ensure that 0-DebugMode is currently selected.

Once you have successfully built an application in debug mode, you are ready to start the application under the control of the Debugger.

You start the Debugger on an application by selecting the application module you wish to execute and selecting Debug from the Project menu or by right clicking on the module name and selecting Debug from the pop-up menu.

Check Function

The Check function was introduced in version 6.1 of PowerCOBOL. It corresponds to the NetCOBOL compiler's "Check" function. It is typically used in debugging mode as it causes additional code to be generated by the compiler and performance is degraded as a result.

You should always rebuild your application when changing the Check option.

The following sub options are available under the Check function when enabled:

Message Count

Sets the number of times certain runtime error messages are to be ignored, thus allowing execution to continue.

Check Item

Selects the target to check.

The Debugger initializes by creating a new Debug tab in the Project Manager window as follows:

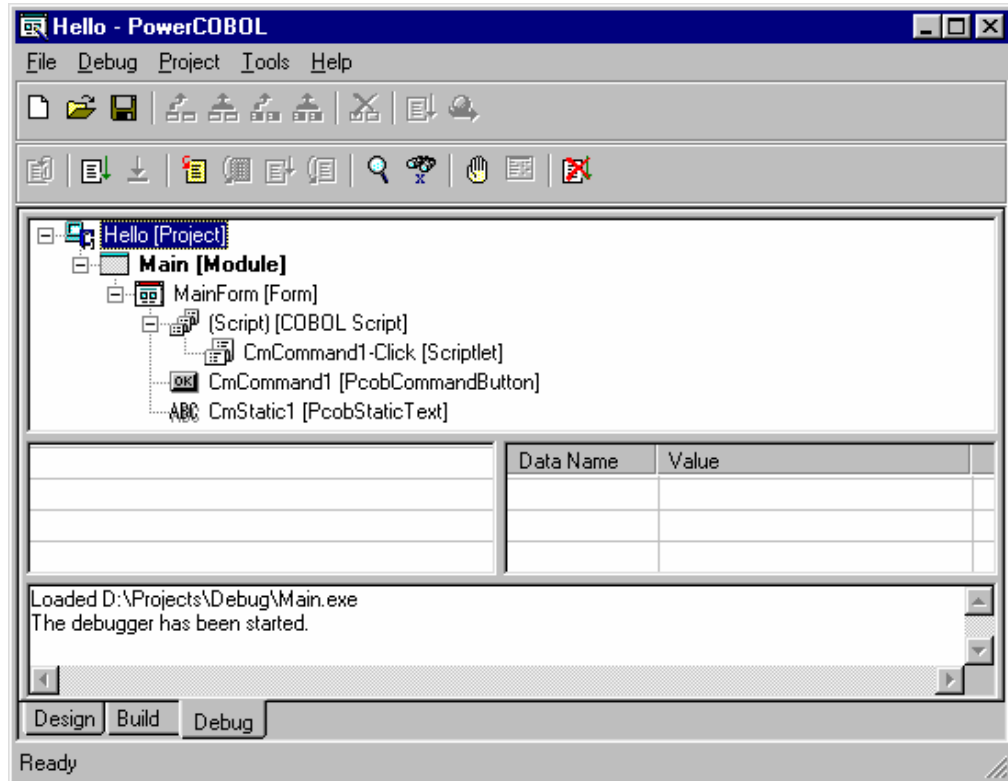


Figure 2.20. The PowerCOBOL Debugger

The Debugger provides a hierarchical tree view of the application's components. It additionally provides an informational window area, a centralized data item watch (monitor) area, and a display area for the current application module and line of code where the execution is currently paused.

You may at any time step between the Debugger and the Build or Design facilities by left clicking the mouse on the appropriate tab near the bottom left corner of the Project Manager window.

This allows you to look at and modify application component properties dynamically during the debug process, for example.

Note that when the initial Debug tab is created as shown in Figure 2.15, the application has not yet begun to execute. This allows you to set initial breakpoints and data items to watch (monitor) before the application begins execution.

Once you are ready to begin execution, you may use either the Go or Step Into options from the Debug menu or click on their associated icons.

The Go option begins executing the application quickly without showing you the source code as it executes. Execution will continue until a breakpoint is encountered or until the user interface gains control and waits for the user to cause an application event.

The Step Into option will display the application's initial source code in a live edit session. You may then step a statement at a time and observe the source code as it executes (you may also select the Go option at any time while stepping).

If you have not specified any initial startup code in your application, the first thing you will typically see is the main form (window) in your application.

See Chapter 7, "Debugging the Program" for more details on using the Debugger.

Chapter 3. Developing Your First PowerCOBOL Application

In this chapter, you will learn how to develop and run a very simple PowerCOBOL application. You will then enhance this simple application by using some of the features in the PowerCOBOL development environment.

Overview

In this exercise, you will develop a simple application with a single graphical user interface (GUI) window. This window will contain a simple text box and a single push button.

When the application first initializes, the text box will contain the value "Begin". When a user clicks on the push button the text in the text box will change to "Hello!".

While this represents the simplest of applications, it will demonstrate the development methodology that you will use when creating PowerCOBOL applications. It will additionally illustrate key features of PowerCOBOL and the natural flow between the tools.

Initializing the PowerCOBOL Development Environment

Start the PowerCOBOL development environment by selecting PowerCOBOL from the Fujitsu NetCOBOL group. From the Start button, select Programs and then select the Fujitsu NetCOBOL for Windows menu. Click on PowerCOBOL.

PowerCOBOL initializes with a project manager. From within this facility, you can open an existing project or use the project wizards to create new projects.

The following window is displayed:

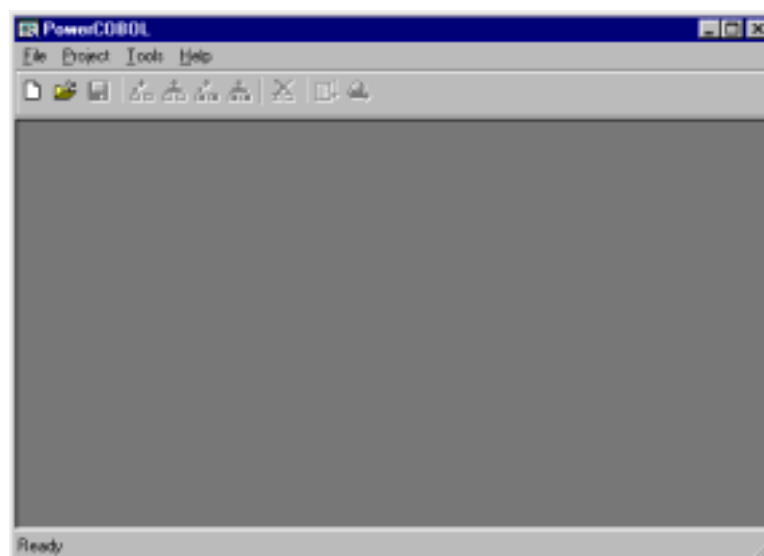


Figure 3.1. The initial PowerCOBOL development window

From the File menu, select New Project. The following window appears:

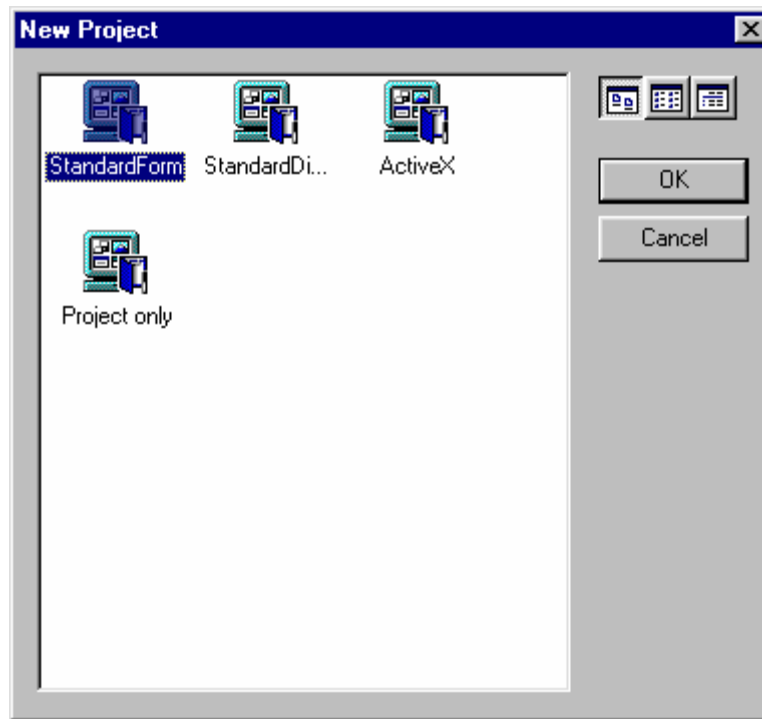


Figure 3.2. The New Project wizard window

Using the mouse, highlight the Standard Form selection in the center and click on the OK button.

The PowerCOBOL new project wizard will now create a template application. This will consist of a form (window) and a few basic properties.

You may use this template application as a starting point to develop GUI applications written in PowerCOBOL.

Note: If you do not want PowerCOBOL to create an application template, you can select the Project only wizard shown in the previous figure.

You will now be presented with the following window:

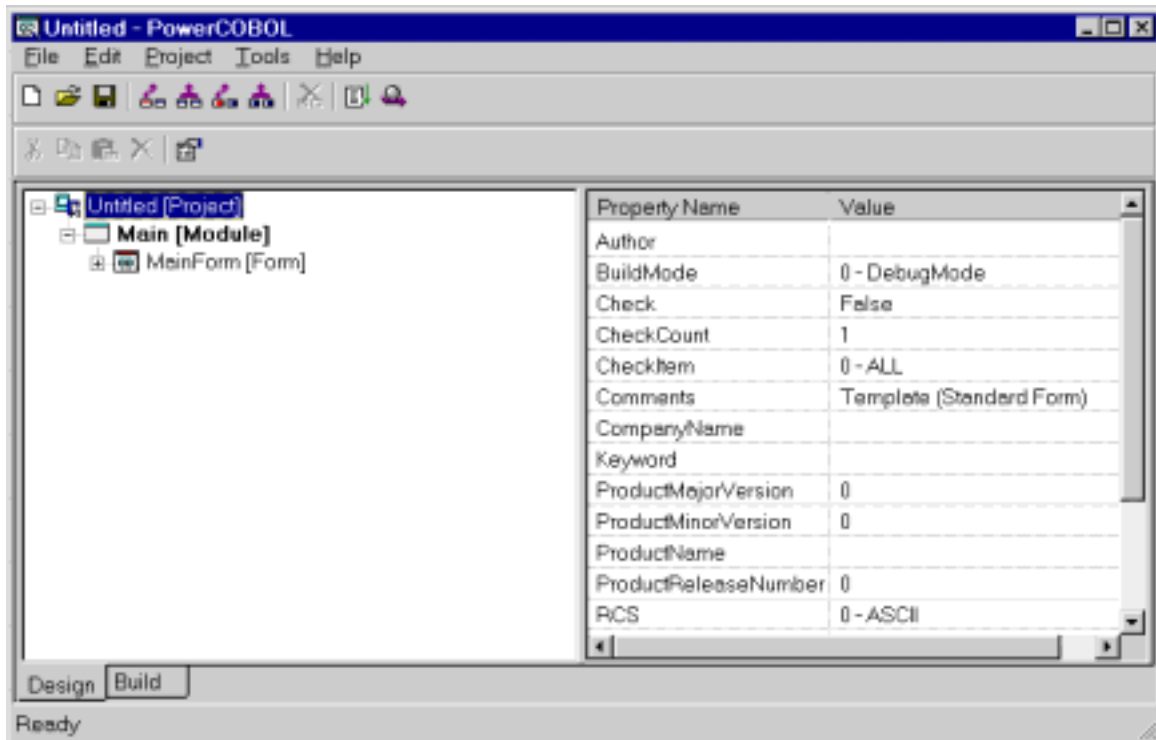


Figure 3.3. The PowerCOBOL Project Manager window

The PowerCOBOL Project Manager window is arranged to show application controls (components) such as forms (windows), dialog boxes, controls (for example, buttons, checkboxes, etc.), and supporting COBOL procedures in the left windowpane.

This information is arranged in a hierarchical tree diagram. You may traverse levels of the hierarchy by clicking on the + and – signs in the tree.

In the right windowpane are the properties associated with the current application control highlighted in the left windowpane. The current values of each property are also displayed.

You may modify a property at any time by clicking on its value and changing it there. In some instances, you may be presented with dropdown menus to select pre-defined properties.

This interface provides a consistent and intuitive mechanism for managing the multitude of properties and possible values associated with GUI development.

You are now ready to begin creating the sample application's graphical user interface.

Developing the Graphical User Interface

In this case, you will have a single application module entitled Main [Module]. This represents the main module of your application. Main is the module that will execute first when you start the application.

Now click on the + sign next to Main [Module]. This will open the top level of components associated with the Main module. The top level will show all forms (windows) associated with this module.

You should have a single form shown and its name should be MainForm[Form]. The main form is the form (window) that will be presented when the application starts.

Go ahead and click on the + sign next to the MainForm[Form] form. This will open up the current list of controls (components and controls) associated with MainForm.

This will show a control named [COBOL Script]. This represents the level at which COBOL Scripts (also called scriptlets or event procedures) will appear. These are small COBOL programs that you write to deal with individual application events that you are interested in. The Project Manager window should now appear as follows:

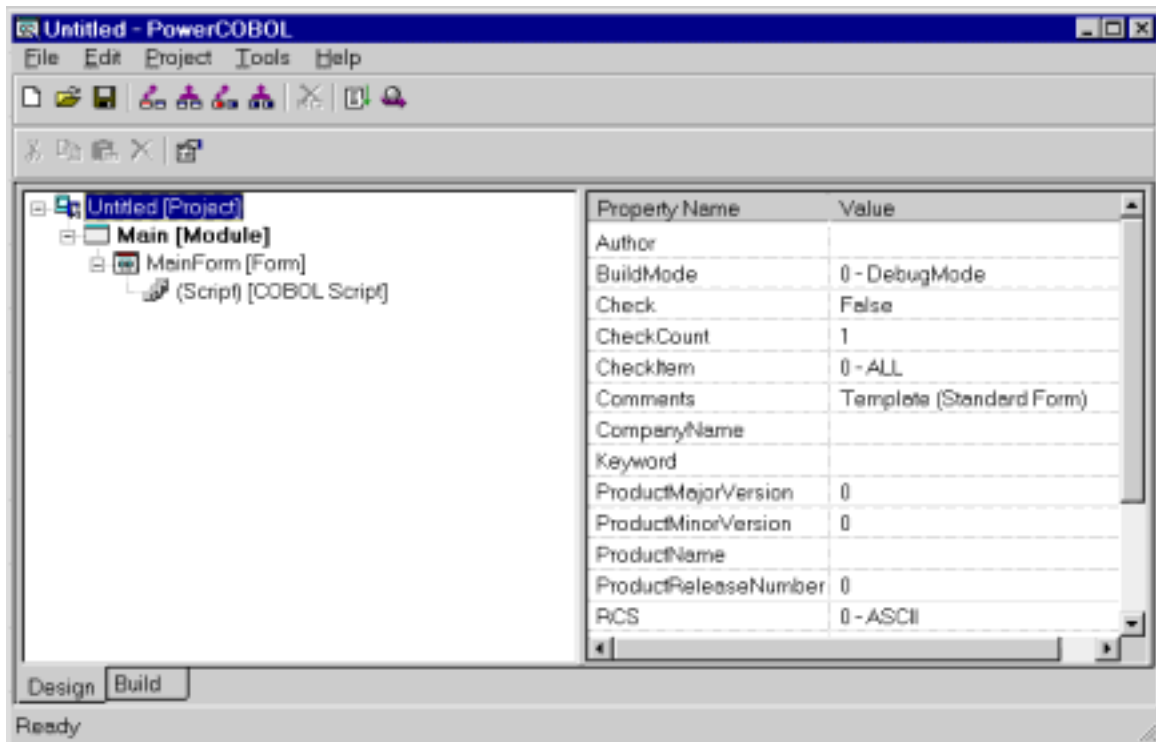


Figure 3.4. The Project Manager window with all application components expanded

Notice that the naming convention in the hierarchical tree uses a name on the left side and the control type in brackets on the right side.

For example, Main[Module] identifies a module named Main, and MainForm[Form] identifies a form (window) named MainForm. MainForm will be our main application window when we execute the project.

Let's begin the development process by changing the title displayed in the top of the MainForm from "MainForm" to "Hello". A form's caption property controls the title of the form when it is displayed on the screen.

In the left windowpane, click on the MainForm[Form] control to highlight it. This will display the properties and their current values for this form in the right windowpane.

In the right windowpane, click on the Caption property value that should currently contain the value MainForm. Change this value to Hello by simply typing over it and pressing the ENTER key.

The caption property has now been changed. It's that simple.

Note that you changed MainForm's title (caption) only - you did not change the object name of MainForm as it's referred to in the development facility.

During the development process you may select any application control's property at any time and modify its initial value in this manner.

You can also dynamically modify the vast majority of these properties during application execution from COBOL procedural code (scripts) that you may choose to create.

For example, if you wanted your form to display initially with a blue background color, you would select the BackColor property and choose blue from the Custom Color palette.

If you instead wanted to wait for some specific event to take place at application run-time to change the background color (such as the user clicking a button), you would do this with a simple COBOL statement at the appropriate script location.

The following chapters will discuss how to write COBOL event procedures to manipulate properties dynamically at application run-time.

For the purpose of this simple application you will modify only the initial properties.

Let's have a look at our MainForm form. Move the mouse to MainForm[Form] in the left windowpane and click on it with the right mouse button to display a pop-up menu.

From the pop-up menu that appears, click on Open. This will display the blank form (window) along with the Form Editor and development tools as follows:

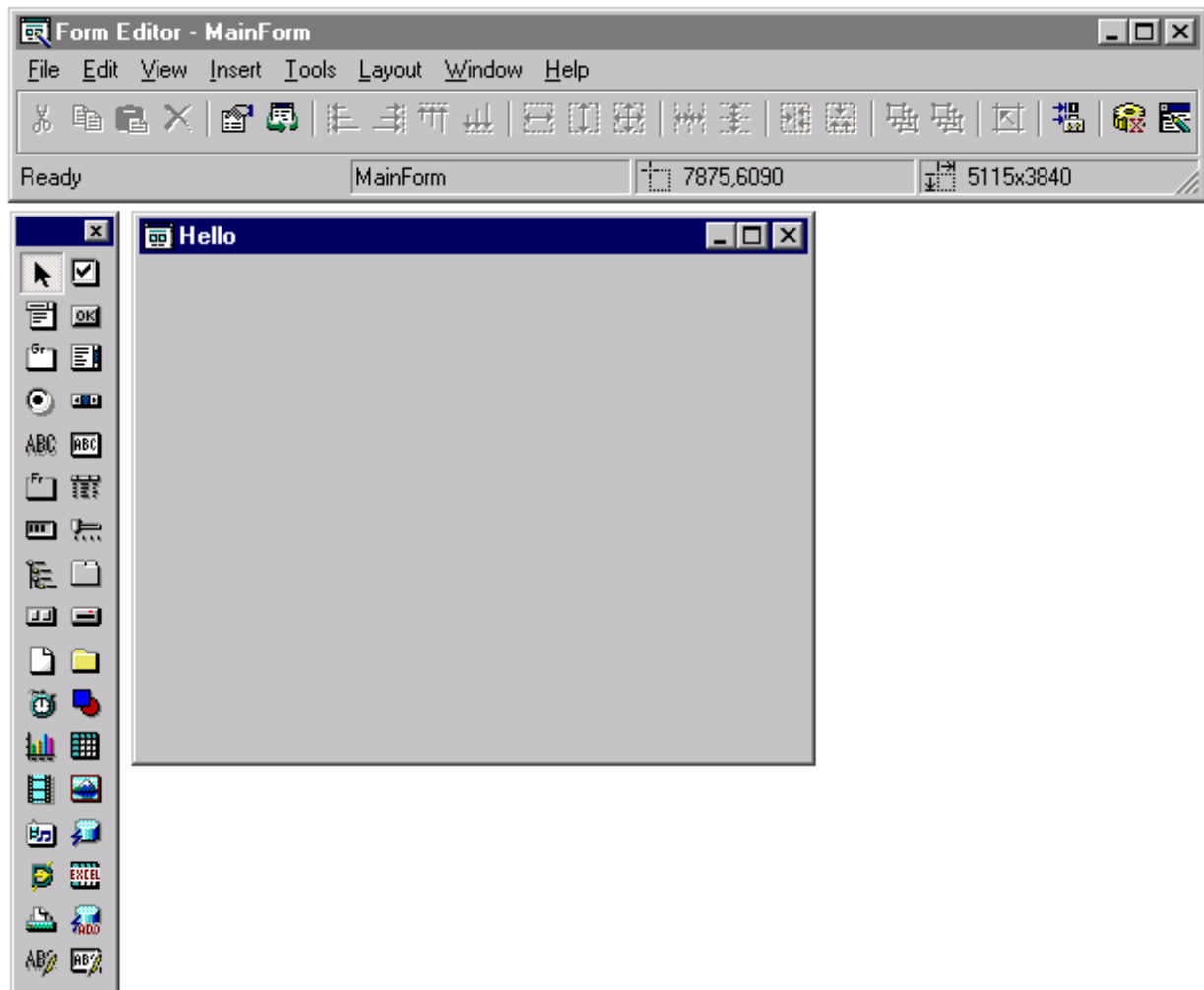



Figure 3.5. The blank MainForm in the Form Editor and the form development tools

Note that the title (caption) of this window is Hello, given the change you made to the value of the caption property.

If the window that appears on your screen does not match the window shown above, you may have selected the wrong project template. If this is the case, go back and close your project and create a new project using the correct project template.

If you examine the small window to the left of the MainForm form, you will note that it represents a toolbox palette. If this palette is not visible on your screen, simply select Toolbox from the View menu.

To develop the Hello application, you will place two of the graphical controls from the Toolbox directly on the Hello application window (MainForm). The first item you will place on MainForm will be a simple push button (called a CommandButton).

Find the PowerCOBOL CommandButton Control icon  in the Toolbox palette and click on it with the left mouse button. This icon is second from the top in the right column. Notice that as you hover the mouse over the toolbox palette controls, a small window (known as a tooltip) will appear stating the name of each control.

Move the mouse over to the MainForm window and position the floating push button near the bottom center of the window. Click on the left mouse button again to paste the push button on the window. If you do not like your initial positioning, simply click on the push button item with the left mouse button, hold it down and drag the button to a new location. You may use this technique at any time to re-position items you place on a form.

The MainForm window should now look something like the following:

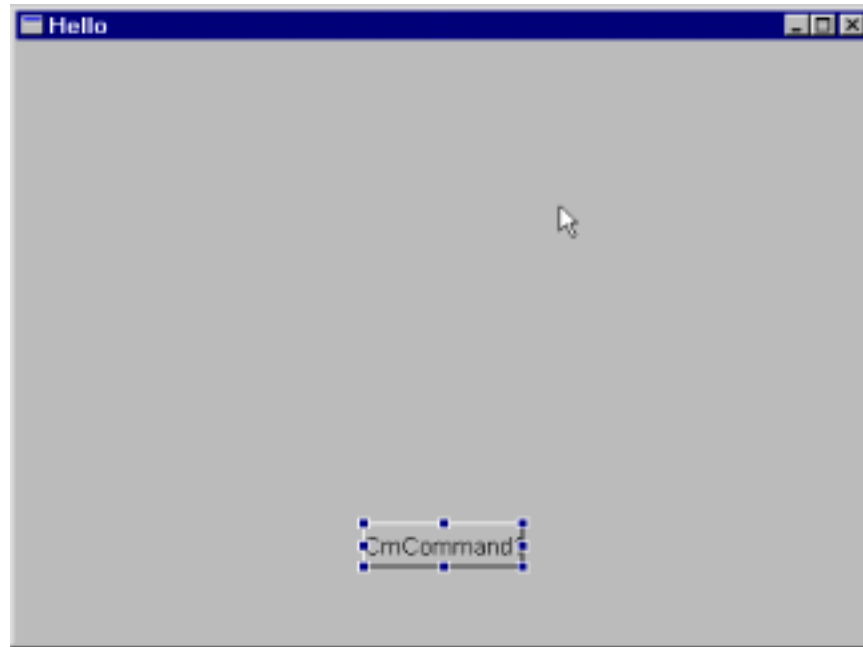


Figure 3.6. The MainForm window with the push button positioned

Notice that this push button is labeled "CmCommand1". PowerCOBOL automatically assigns default names and captions (where appropriate) to controls as you create them. You can change the caption or the actual control name at any time. In most cases the default caption and the control name will be identical. They do not have to be identical however.

You will now change the text displayed on the push button from "CmCommand1" to "Hello". Note that this will only change the caption of the push button. It will not change the name of the push button, which will remain as "CmCommand1".

You accomplish this by bringing up the Properties dialog box for this control. You can modify properties for most graphical controls in the manner described below.

Move the mouse over the push button and right-click once to display a pop-up menu.

At the bottom of the pop-up menu, click on Properties. This will display the following Properties dialog box:

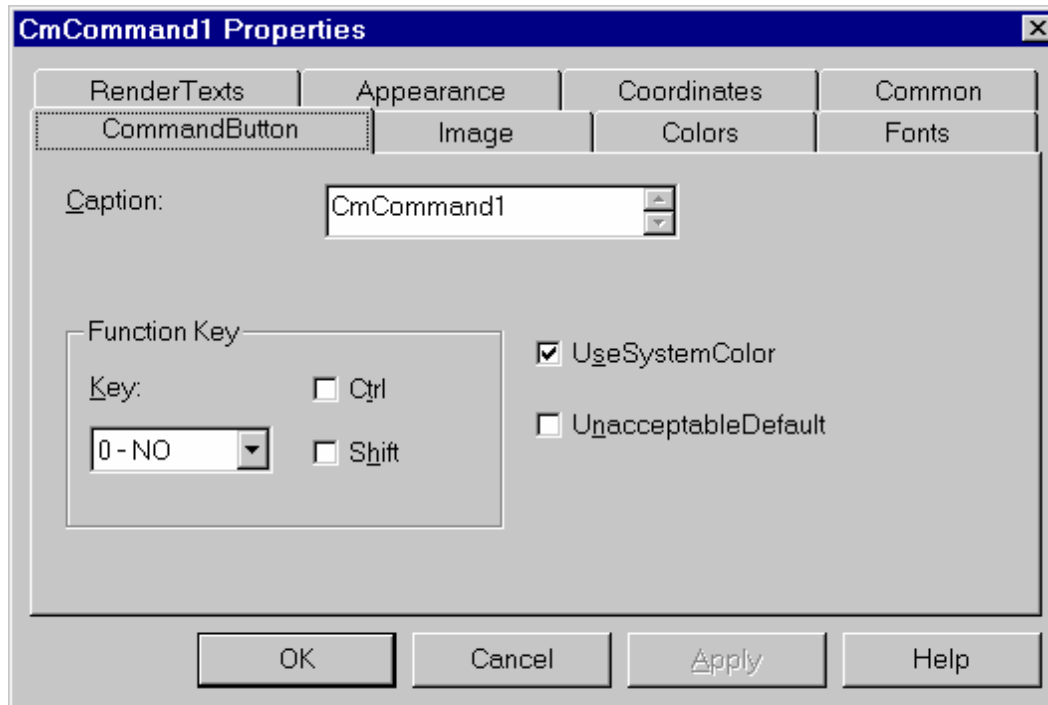



Figure 3.7. The Properties dialog box for the push button control

The text displayed on the push button is known as its caption. Select the Caption field in the Properties dialog box and change "CmCommand1" to "Hello".

Now click on the OK button.

The push button on MainForm should now be labeled "Hello".

You will now place a static text control  on MainForm. Move the mouse back to the Toolbox palette and click the left mouse button on the PowerCOBOL Static Text control icon. This icon is fifth down from the top in the left column.

Now move the mouse back over to MainForm and position the floating text box control near the top center of the form. Click on the left mouse button again to paste the static text control on the form.

Note that the text string "CmStatic1" is displayed in the new static text control and that it is located in the upper left corner of this control.

You will change this by bringing up the Properties dialog box for the static text control. Move the mouse over the static text control and right-click once to display a pop-up menu.

Select Properties from the bottom of the pop-up menu by clicking on it. The following Properties dialog box will appear:

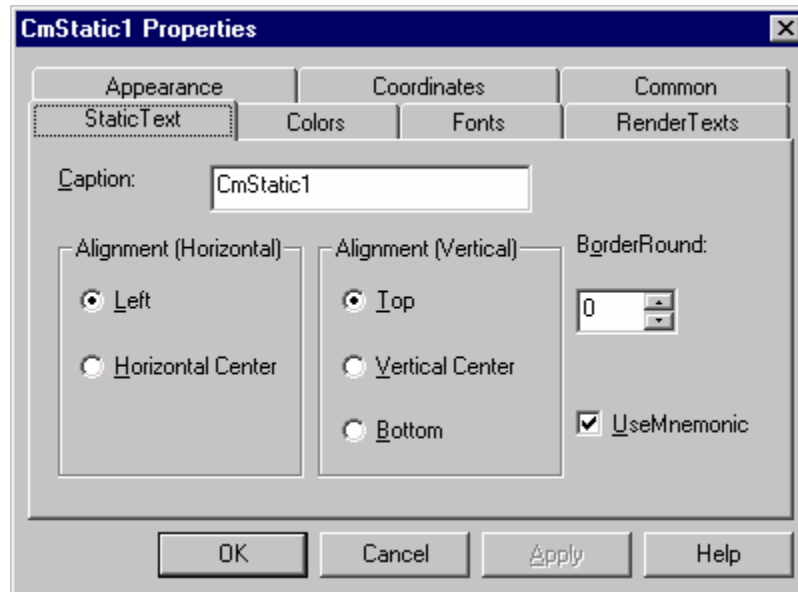


Figure 3.8. The Properties dialog box for the text control

Move to the Caption field and change "CmStatic1" to "Begin".

Additionally, it would be nice to center any text displayed in this static text control. You can do this in the Properties dialog box by clicking on the Horiz. Center and Vert. Center check boxes. Do this now.

Now click the OK button. The text displayed should now be "Begin", and it should be nicely centered within the static text control.

You have now completed the 'painting' of the application. The MainForm window should now look something like the following:



Figure 3.9. The MainForm application window after painting

You are now ready to write the event procedures (COBOL Scripts) to complete the application.

Writing the Event Procedures

You are now ready to complete the Hello application. PowerCOBOL will generate and manage the GUI portion of this application. All you need to do now is to define any specific behavior you desire within this application that PowerCOBOL will not automatically manage.

This means that you need to create the behavior needed to ensure that when a user selects the Hello push button, the text string in the text box control will change from "Begin" to "Hello!".

In many other graphical development environments, figuring out just how to do this and deciding where to place the program code can be quite challenging. However, PowerCOBOL makes this quite simple for you.

The programming technique that PowerCOBOL uses is widely known as *event-driven programming*. That is, certain program functions are invoked whenever a specific event occurs.

In the case of the Hello application, you need to define behavior for one specific event - when a user selects the Hello push button (actually known as a `CommandButton` control, which in this case is named "CmCommand1") by clicking on it with the left mouse button.

In order to define this behavior in the correct place in your application, move the mouse over the CmCommand1 `CommandButton` control (the push button labeled "Hello") and right-click once to display a pop-up menu.

Now move the mouse over the field entitled Edit Event Procedure. Do not click on the mouse button yet. Instead, you should notice a secondary pop-up menu.

This secondary pop-up menu identifies all of the possible actions (events) associated with the current control (in this case, the CmCommand1 `CommandButton` control).

Move the mouse over the event entitled Click and select it by left clicking on it. This will display the PowerCOBOL Editor to create or modify COBOL code associated with the Click event of the CmCommand1 push button.

Although you previously changed the caption of the CmCommand1 push button to "Hello", the actual name of this push button is still "CmCommand1" as assigned by PowerCOBOL when it was created.

You may now use the Editor to create COBOL code for this specific event. At run-time, each time a user clicks on the push button labeled Hello, this COBOL code will be executed.

PowerCOBOL will automatically handle every possible event in a PowerCOBOL application in some default fashion. It is only when you need to define behavior for a specific event that you need to write an event procedure.

In the case of the Click event for the CmCommand1 push button control (which you've labeled Hello), you need to add the following line of code to the `PROCEDURE DIVISION` of its associated event procedure. Make sure you start the line below in column 12, just under the `PROCEDURE DIVISION` statement.

```
Move "Hello!" To "Caption" Of CmStatic1.
```

Note that CmStatic1 is the name of the text box control (the object on the MainForm window that text will be displayed within). When the Hello push button is selected, you want the text "Begin" to change to "Hello".

The single line of code above instructs PowerCOBOL to place "Hello" in the caption (text display) of the CmStatic1 text control. Because you have placed this line of code in the event procedure for the Click event of the CmCommand1 push button, this statement will be executed every time the push button labeled "Hello" is clicked on at run-time.

When you have completed this edit task, the Editor window should look like the following (note that the screen shots of the PowerCOBOL editor used in this manual have the Editor's toolbar turned off, so your Editor may appear differently):

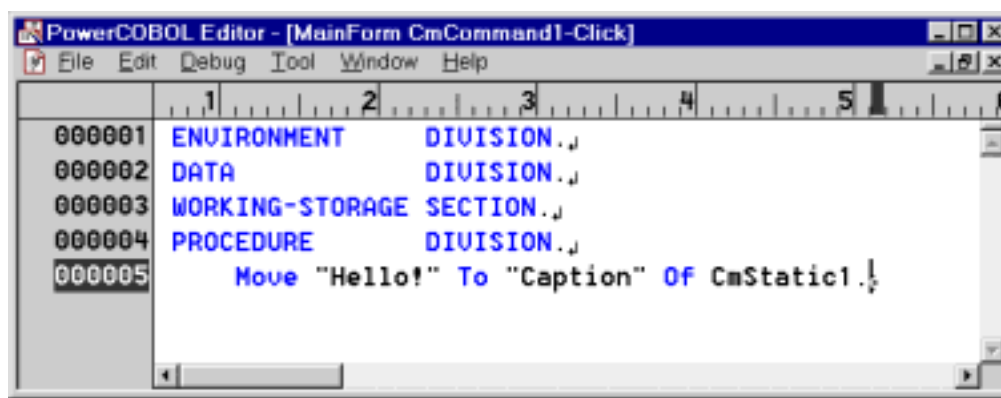


Figure 3.10. The code for the Click event of the CmCommand1 push button

Notice in Figure 3.10 that there is a carriage return symbol at the end of each line. If you wish to turn this off, simply select Options from the Tool menu and uncheck the Display Returns check box, which will be done for the remainder of this manual.

Once you have completed entering the line of code in the appropriate position, save the event procedure code by selecting Save from the File menu. Then close the Editor.

Now close the MainForm window that you've created by selecting either Close or Exit and return to shell from the File menu in the Form Editor window.

The work you've just done on the MainForm window will be saved automatically.

You have now completed the initial development of the Hello application. It's time to compile and link the application together using the PowerCOBOL Build facility.

Building and Running the Application

You are ready to compile, link, and run the Hello application. PowerCOBOL makes this a very simple and straightforward process.

Before you build your application, however, you may want to make a few simple modifications to the project.

Go back to the main Project Manager window, which shows your project controls and their current properties, and expand all of the controls.

You can accomplish this by moving the mouse over the Untitled[Project] control and right clicking the mouse to display a pop-up menu. Then select Expand All.

Your project should look something like the following:

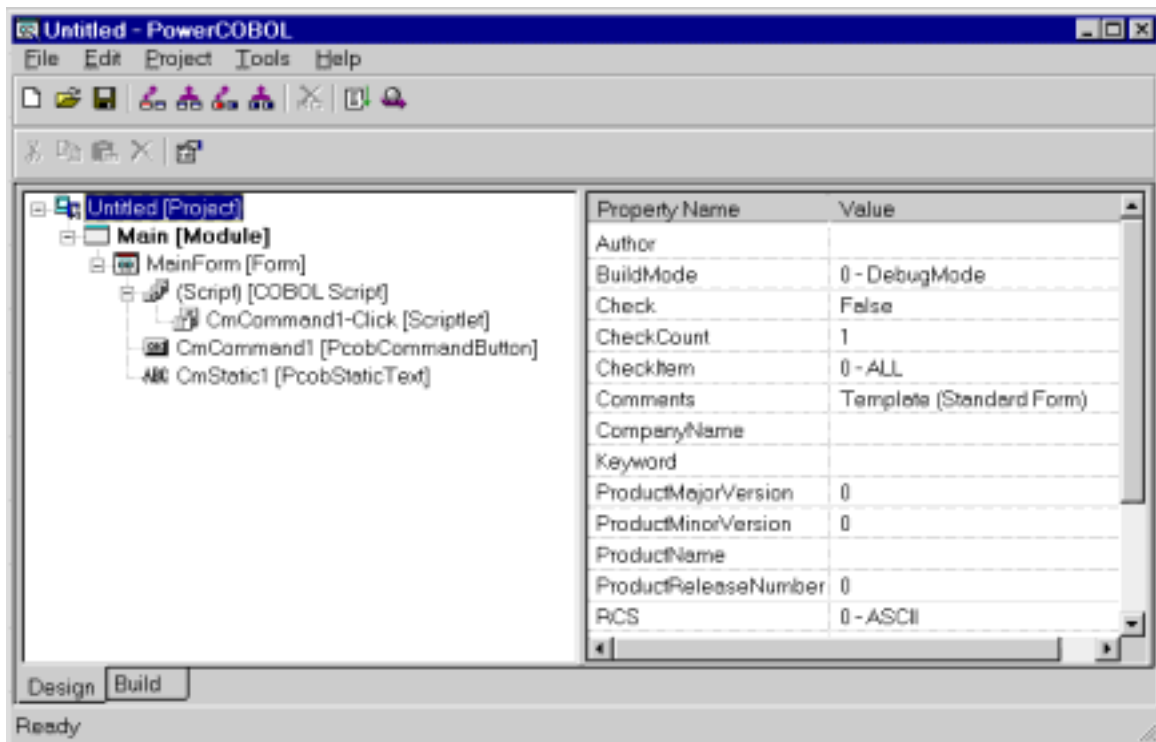


Figure 3.11. The Project Manager window with all controls expanded

From this window, you may select any application control and change its initial properties. You may also edit and change any of the application controls.

For example, if you needed to add another push button control to the MainForm window, you can simply move the mouse over the MainForm name in the left windowpane and right-click once to display a pop-up menu.

From this pop-up menu, you could select Open to display the form (window) in the Form Editor. You may typically access any of the application controls or their associated event procedures (COBOL scripts) in this manner. If you decide you need to edit one of the COBOL scripts, you may right click on it in the Project Manager window and select the Edit option from the context menu that appears.

Additionally, you may wish to change the pre-assigned names given to the application controls by PowerCOBOL.

For example, if you examine the application project window for the Hello application, you will note that the application module (executable file) name is Main.

This means that if you build this application, the resulting executable (.EXE) file will be named Main.exe. A more meaningful name for this file would be Hello.exe.

To change the name of the executable module, move the mouse over Main[Module] in the left windowpane and right-click once with the mouse to display a pop-up menu.

Now select Rename near the bottom of the pop-up menu. This will open up the name field for modification. Change "Main" to "Hello" and press the ENTER key.

You may generally change the names of any of the application controls in this manner.

Use this technique to change the name of MainForm[Form] to Hello. Notice that this may also invoke a dialog box asking you if you wish to change "MainForm" to "Hello" everywhere it is referenced in the project. If so, respond yes, and the global find and change facility will be invoked and the change will take place automatically. This prevents you from having to manually hunt down any COBOL statement that refers to the old name and change it.

You should also change the name of this project from "Untitled" to "Hello". If you attempt to use the same technique, however, you will note that the pop-up menu for the project name does not offer a Rename option.

Instead, you must go to the File menu and select Save As to save the project under a different name. This will then automatically change the project name to the new name.

Go ahead and save your project using a name of Hello.

You are now ready to build the Hello application.

Move the mouse over the Hello[Project] control in the left windowpane and right-click once to display a pop-up menu. Select All Build.

The Hello application should be created, compiled and linked without errors. If so, the Build tab will be exited and you will be placed back in the main Project Manager window. Click on the Build tab near the bottom left of the Project Manager to view your results.

You should see results that look something like the following:

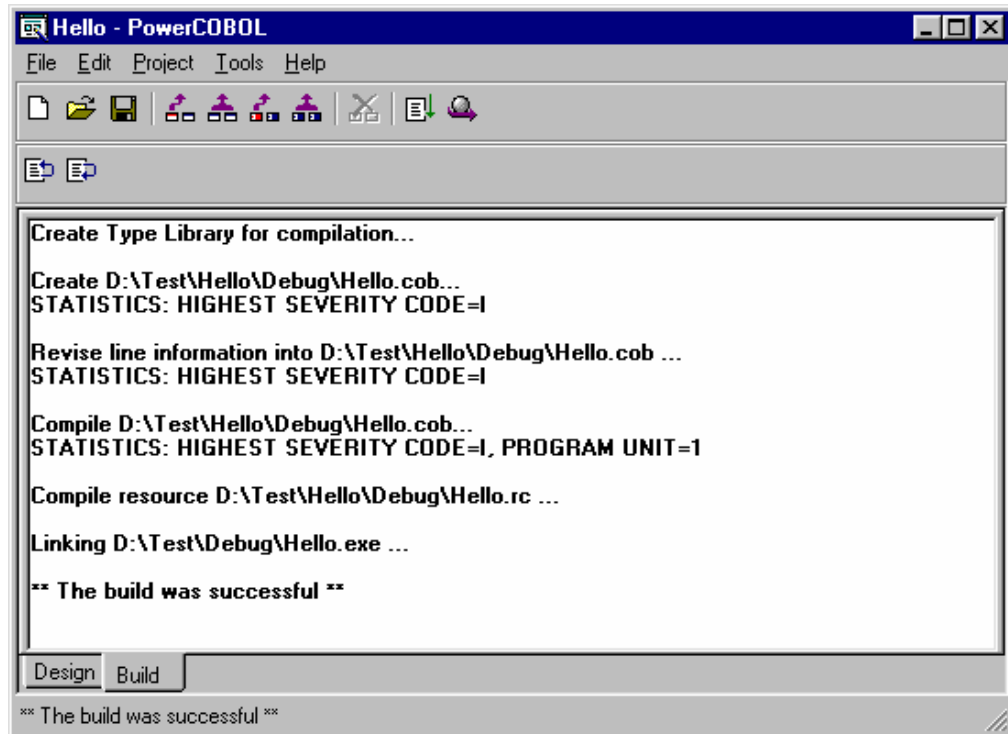


Figure 3.12. Results of a successful build of the Hello application

If you receive any errors, the Build tab will not disappear. You can use Next Error and Previous Error in the Edit menu (or click on their corresponding icons) to position you at the point the error occurred in your application code. You can also simply double click the left mouse button on any error text that appears in red and you will be positioned in the PowerCOBOL editor at that point in your code. If you received any errors, correct them and build the project again until there are no errors.

Once you have a successful build and wish to run the Hello application, move the mouse over the Hello[Module] control in the left windowpane and right-click once to display a pop-up menu.

Now select Execute.

The following application window should appear on your desktop:



Figure 3.13. The Hello application window

If you click on the Hello push button, the text value displayed in the text box control should change from "Begin" to "Hello!".

To exit the application, click on the close (X) button in the upper right corner of the application window.

Congratulations, you've just created your first PowerCOBOL application.

If you are curious to see the actual COBOL program created by PowerCOBOL for this application (remember - you only had to enter a single line of COBOL code!), you should first be aware that PowerCOBOL first generates a high-level source program known as a .PRC file. This .PRC file is analyzed by the PowerCOBOL preprocessor that creates an actual COBOL program (.COB file). You should never have to look at either of these files as PowerCOBOL manages them for you. If you are curious as to what the .PRC file looks like for the Hello application you just built, here it is:

```
#FILE D:\Test\Hello\Debug\Hello.PRC
000001 IDENTIFICATION DIVISION.
000002* Hello.
000003 PROGRAM-ID.      Hello.
000004 ENVIRONMENT      DIVISION.
000005 CONFIGURATION    SECTION.
#LINE 12
000012 SPECIAL-NAMES.
000013 REPOSITORY.
000014 .
000015 INPUT-OUTPUT      SECTION.
000016 FILE-CONTROL.
000017 DATA              DIVISION.
#LINE 22
000022 LINKAGE          SECTION.
000023 01  POW-FORM IS GLOBAL.
000024 02  POW-SELF PIC S9(9) COMP-5.
000025 02  POW-SUPER PIC X(4).
000026 02  POW-THIS PIC S9(9) COMP-5.
```

```

000027 02 CmCommand1 PIC S9(9) COMP-5.
000028 02 CmStatic1 PIC S9(9) COMP-5.
000029 01 Hello REDEFINES POW-FORM GLOBAL PIC S9(9) COMP-5.
000030 01 POW-CONTROL-ID PIC S9(9) COMP-5.
000031 01 POW-EVENT-ID PIC S9(9) COMP-5.
000032 01 POW-OLE-PARAM PIC X(4).
000033 01 POW-OLE-RETURN PIC X(4).
000034 PROCEDURE DIVISION USING POW-FORM POW-CONTROL-ID POW-EVENT-ID
POW-OLE-PARAM POW-OLE-RETURN.
000035 EVALUATE POW-CONTROL-ID
000036 WHEN 117440514
000037 EVALUATE POW-EVENT-ID
000038 WHEN -600
000039 CALL "POW-SCRIPTLET1"
000040 END-EVALUATE
000041 END-EVALUATE
000042 EXIT PROGRAM.
000043 IDENTIFICATION DIVISION.
000044* CmCommand1-Click.
000045 PROGRAM-ID. POW-SCRIPTLET1.
000046*<SCRIPT DIVISION="PROCEDURE", CONTROL="CmCommand1", EVENT="Click",
POW-NAME="SCRIPTLET1", TYPE="ETC">
000047 ENVIRONMENT DIVISION.
000048 DATA DIVISION.
000049 WORKING-STORAGE SECTION.
#LINE 50,#START,#OTHER
000050 01 POW-0000 PIC S9(18) COMP-5.
000050 01 POW-0001 PIC S9(9) COMP-5.
000050 01 POW-0002 PIC S9(9) COMP-5.
000050 01 POW-0003 PIC S9(9) COMP-5.
000050 01 POW-0004 PIC S9(9) COMP-5.
000050 01 POW-0005 PIC S9(9) COMP-5.
000050 01 POW-0006 PIC X(8192).
000050 01 POW-0007 PIC S9(9) COMP-5.
#LINE 49,#END
000050 PROCEDURE DIVISION.
000051 MOVE -518 TO POW-0000
#LINE 51,#START,#OTHER(51,39)
000051 MOVE 4 TO POW-0001
000051 MOVE 0 TO POW-0002
000051 MOVE 0 TO POW-0003
000051 MOVE 0 TO POW-0004
000051 MOVE 1 TO POW-0005
000051 MOVE "Hello!" TO POW-0006
000051 MOVE 33636360 TO POW-0007
000051 CALL "XPOW_INVOKE_BY_ID_2" USING VALUE CMSTATIC1 REFERENCE POW-
0000
000051 VALUE POW-0001 POW-0002 POW-0003 POW-0004 POW-0005 POW-0007
000051 REFERENCE POW-0006 END-CALL
000051
#LINE 51,#END
000052*</SCRIPT>
000053 END PROGRAM POW-SCRIPTLET1.
000054 END PROGRAM Hello.
#FILE

```


If you see a few unfamiliar statements in the above noted COBOL source code, please note the following about PowerCOBOL:

- PowerCOBOL produces a high level source code file with a .PRC extension that is input into a preprocessor which then expands it into an actual COBOL program (.COB File).
- The first line of the above listing contains the name of the .PRC file that was input into the PowerCOBOL preprocessor (Hello.prc).
- PowerCOBOL makes extensive use of the ANSI 85 update to the COBOL language regarding nested COBOL programs. You will note that the event procedure you wrote (COBOL script) to handle the Click even for the command button appears as a nested COBOL program named "POW-Scriptlet1" beginning on line number 45.
- The #LINE statements are a Fujitsu preprocessor extension. These identify which lines in your original COBOL scriptlets match up to the expanded program listing for purposes of debugging. This allows the debugger to synchronize the original program listing with the expanded code output by the preprocessor transparently. The result of this is that you actually debug your original source code in the debugger, not even realizing that certain statements may have been expanded and/or commented out and replaced by additional code.

The real beauty is that you don't have to deal with writing all of this program code - PowerCOBOL does it all for you!

In the next section you will enhance the Hello application to illustrate a few of the very powerful and easy-to-use functions in PowerCOBOL.

Enhancing the Application

You will now add a few minor enhancements to the Hello application. The first two enhancements will center on changing fonts and color. The third enhancement will add a bit of program code to create more flexible behavior when a user clicks on the Hello push button.

If you have not quit running the Hello application, do so now by clicking on the close (X) button in the upper right corner of the Hello application window.

Make sure you are back at the Project Manager window, which should now look something like the following:

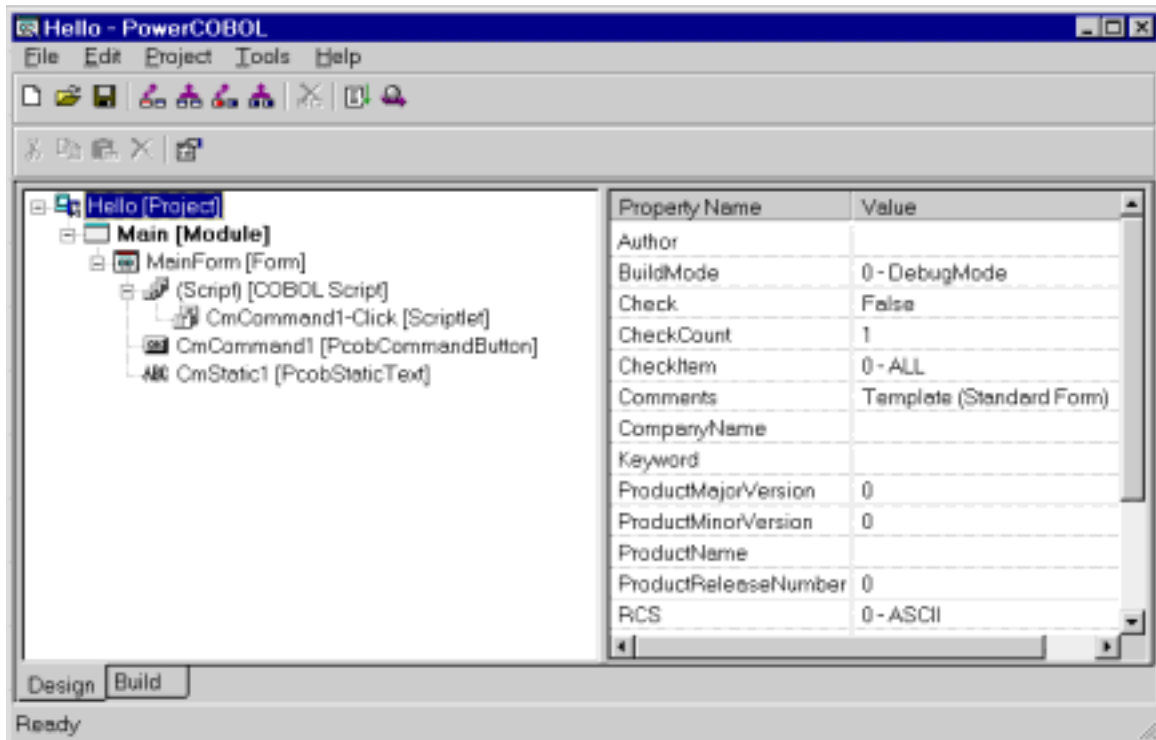


Figure 3.14. The Hello application in the Project Manager window

Enlarging the Font in the Text Box Control

In reviewing the current Hello application, you may note that the size of the text in the text box control is quite small compared to the size of the actual area it resides in. If you previously resized this box to more closely match the text size, resize it back in order to enlarge it again.

PowerCOBOL allows you to easily change the font, the font size, or the font style for most controls that can be displayed in a PowerCOBOL form (window).

Select the CmStatic1[PcobStaticText] text box control by moving the mouse to it and left clicking once on it. This will display the current properties for this text box control where the application displays "Begin" and "Hello!".

Move the mouse to the Font property in the right windowpane. Click once on Font. This will highlight the Font property and you will notice a small push button to the far right.

This push button indicates that there are multiple predefined and selectable options for this specific property. Click on the push button.

This will display the Font Properties as follows:



Figure 3.15. The Font Properties dialog box

The fonts available to the application will depend on which fonts are installed on the machine where the application is currently running. In the case of the above box, the font is MS Sans Serif and its size is 8.25 points.

You may change the font type, size, and style using this dialog box. Go ahead and change the font size to 18 points and the Style to Italic. Click on the OK button.

That's all there is to it when you wish to change a font type, font size or font style! You can even change font attributes at run-time by adding the appropriate program code to your event procedure, which you will do later in this chapter.

Changing the Color of an Item

You will now change the color of the text displayed in the text box control named CmStatic1[PcobStaticText].

Select the ForeColor (foreground color) property by left clicking once on it. Now click on the small push button that appears to the far right.

This will display the ForeColor Properties dialog box as follows:

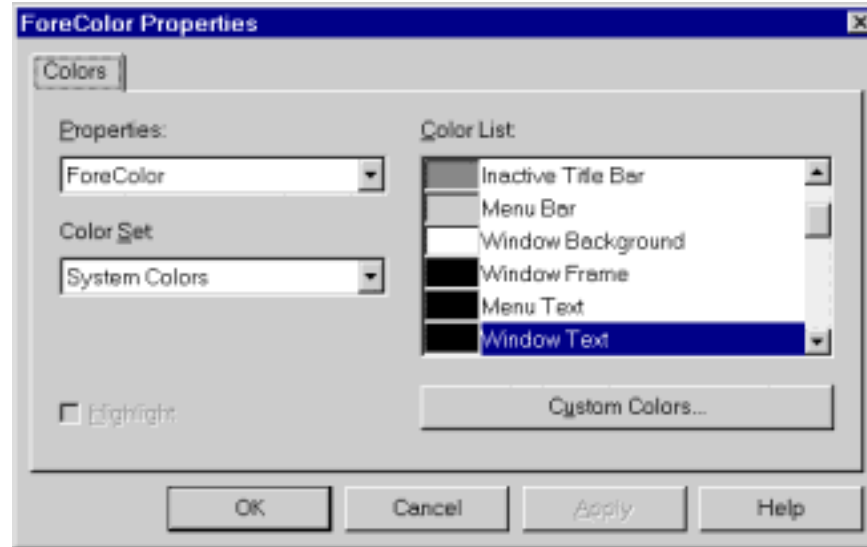


Figure 3.16. The ForeColor Properties dialog box

Under ColorSet, select Standard Color. This will display standard colors to select from. Choose a color you like, such as blue, and click on the OK button.

That's all there is to it when you need to change colors of text-oriented fields.

Enhancing the Push Button Behavior

You may have noticed that when you execute the Hello application and click on the Hello push button, the text string in text box control changes from "Begin" to "Hello!" as expected.

However, if you click on the Hello button again, nothing seems to happen. Actually, something is happening - the value of the text string is being set to "Hello!" – it's just that it is constantly being set to the same value each time so you don't see any visual effect on the desktop.

You will now alter this behavior to toggle the text value each time a user clicks on the Hello push button.

You do this by adding additional program code to the event procedure for the Hello push button (remember that "CmCommand1" is the actual PowerCOBOL control name for the one and only push button you have created, while its text value is Hello).

To add COBOL code for the desired behavior, move the mouse over the CmCommand1[PcobCommandButton] push button control and right-click once to display a pop-up menu.

Move the mouse to Edit Event Procedure on the pop-up menu to display a secondary pop-up menu. Now move the mouse to the *Click event and left-click once on it. (The '*' in front of the event name indicates that you have previously created custom procedure code for this event).

An even simpler way to edit an existing COBOL scriptlet is to simply move the mouse over it in the Project Manager window, right click on it and left click on Edit in the context menu that appears.

This will display the PowerCOBOL Editor containing the procedural code you created earlier:

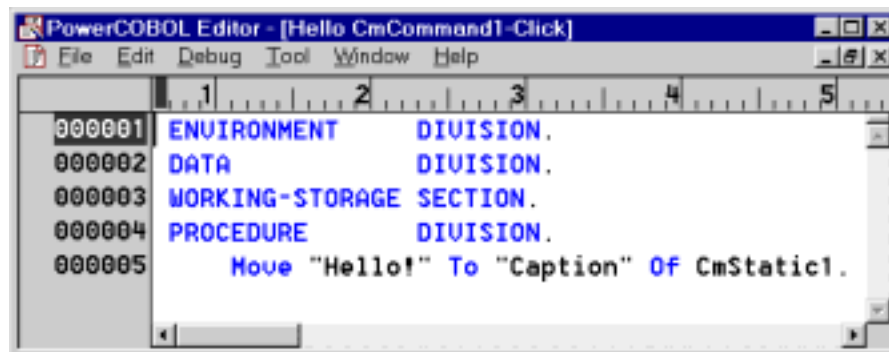


Figure 3.17. Current event procedure for the CmCommand1 Click event

You will now add additional code to this event procedure. The behavior you want to achieve is to toggle the text value in the text box control each time the user clicks on the Hello push button.

In order to achieve this behavior, you will need to define a data item to contain the current state of the text box, and add some procedural code to implement the desired behavior to toggle the text value.

Because you are dealing with COBOL, this is relatively straightforward. Modify the event code so that the event procedure in the CmCommand1 Click event window appears as follows:

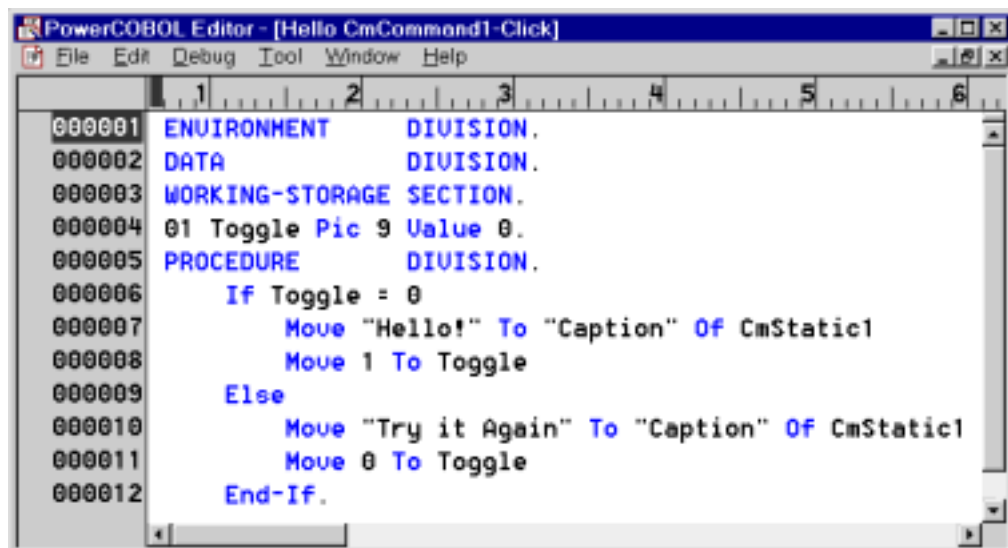


Figure 3.18. The updated event procedure for the Hello Click event

Save the code and close the Editor window by clicking on the close (X) button in the upper right corner of the window.

Now save your project work by selecting Save in the File Menu.

You are now ready to try out your new enhancements. Move the mouse over the Hello[Project] control and right-click once to display a pop-up menu.

Now select Rebuild All. Your program will be recompiled and re-linked into an executable file. If you encounter any errors, carefully retrace the steps above.

You are now ready to run the enhanced Hello application. Move the mouse over the Hello[Module] control and right-click once on it to display a pop-up menu. Now select Execute.

The Hello application will appear on your desktop. The new font size and color scheme should be apparent in the text control. When you click on the Hello push button, you should experience the new behavior from the toggle code you placed in the application.

You may, however, notice a new problem. The actual text control may be too small to display the larger font size you selected, and may chop it off.

If this is the case, you need to enlarge the size of the text control area (CmStatic1).

You could accomplish this by modifying the Height and Width properties of the CmStatic1 control in the right windowpane. Chances are, however, that you would not know the precise numbers to place in the Height and Width properties.

It is a better bet that you could more easily select the correct size by actually viewing and enlarging the text control in the form edit mode.

Close the application by selecting the close (X) button in the upper right corner of the window. Go back into the Form Editor by moving the mouse over the Hello[Form] control and right clicking on it to display a pop-up menu. Click on Open.

This will display the Hello application form (window) in the Form Editor.

Left-click once on the CmStatic1 StaticText control to select it (you should notice a series of small square dots outlining this text box). Now simply select one of the horizontal sides by clicking and holding the left mouse button down and dragging it outwards. Now do the same for one of the vertical sides.

Drag the sides until you are convinced the text box is large enough to hold the "Try it Again" text string.

Now close the Form Editor by selecting Exit from the File menu or by clicking on the close icon (X) in the upper right corner. Notice you don't have to save the form before exiting the Form Editor.

Now save your project and rebuild it. Execute it to convince yourself you have corrected the problem.

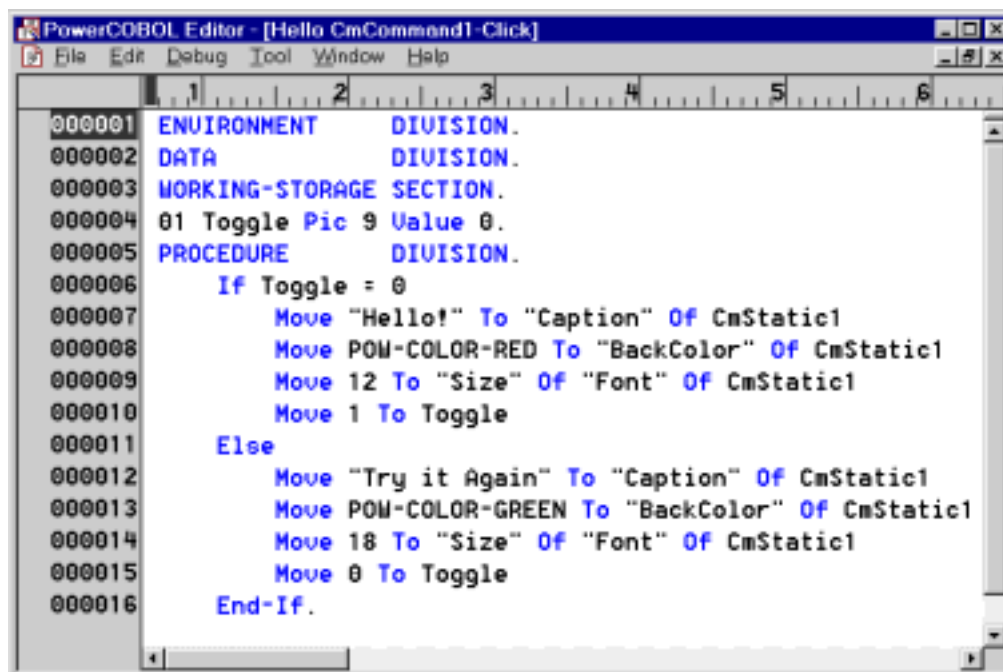
When you have finished experimenting with this new version of the Hello application, quit the application by clicking on the close (X) button in the upper right corner of the window.

Adding Support to Change the Font Size and Color at Run-time

You will now make a few simple changes to the event procedure for the Hello push button to change the font size and color in the CmStatic1 text control.

Move the mouse over the CmCommand1-Click[Scriptlet] in the left windowpane and right-click once to display a pop-up menu. Select the Edit option from the context menu that appears to bring up the PowerCOBOL editor on this event procedure.

This displays an edit session for the Click event. Alter the code as follows:



```

000001 ENVIRONMENT      DIVISION.
000002 DATA              DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Toggle Pic 9 Value 0.
000005 PROCEDURE         DIVISION.
000006     If Toggle = 0
000007         Move "Hello!" To "Caption" Of CmStatic1
000008         Move POW-COLOR-RED To "BackColor" Of CmStatic1
000009         Move 12 To "Size" Of "Font" Of CmStatic1
000010         Move 1 To Toggle
000011     Else
000012         Move "Try it Again" To "Caption" Of CmStatic1
000013         Move POW-COLOR-GREEN To "BackColor" Of CmStatic1
000014         Move 18 To "Size" Of "Font" Of CmStatic1
000015         Move 0 To Toggle
000016     End-If.
  
```

Figure 3.19. The Hello push button event procedure code with font and color changes

The code shown in Figure 3.19 also illustrates two important points:

1. Some PowerCOBOL properties, such as "Font", contain sub-properties, such as "Size". When moving a value to or from such a property it is thus reference such as:

"Size" Of "Font"

Use the PowerCOBOL Help system to look up properties and verify if they have sub-properties. You will receive a compile error if you attempt to move a value to a property such as "Font" that contains sub-properties.

2. PowerCOBOL provides a number of system constants such as POW-COLOR-GREEN to make your programming tasks easier. Use the PowerCOBOL help system to look up these constants as needed or simply to view all of the constants available to you.

Save this code change and close the Editor. Now save the project and rebuild it. Execute the application.

Feel free to experiment with the application. If you click multiple times on the Hello push button, you will notice that the text value, background color, and font size all change upon each click. You may also notice that the text does not fit nicely in the static text control. You can change this by decreasing the font size you selected, or by making the control larger.

When you are finished experimenting, quit the application by clicking on the close (X) button in the upper right corner of the window.

Further Enhancing the Application

In this section you are going to add a few more enhancements to the Hello application.

You will first rearrange the form (window) to make room for the new items you are going to add. Make sure you have the Hello form open in the Form Editor.

You will then add three new controls to the form, which relate to file names. These include a DriveList control, a FolderList control, and a FileList control.

This set of enhancements will illustrate some cooperative behavior between different controls. You will create a facility to allow you to browse the names of files on your system and to select one at any time.

Rearranging the Current Application Window (Form)

You are going to move the static text control (CmStatic1) toward the bottom of the form just above the Hello push button to make room for the new items you are going to add. You can resize the form if needed as well.

Use the mouse to select and drag the CmStatic1 text control towards the bottom of the form so that it is just above the Hello push button.

Your form should now look something like the following:

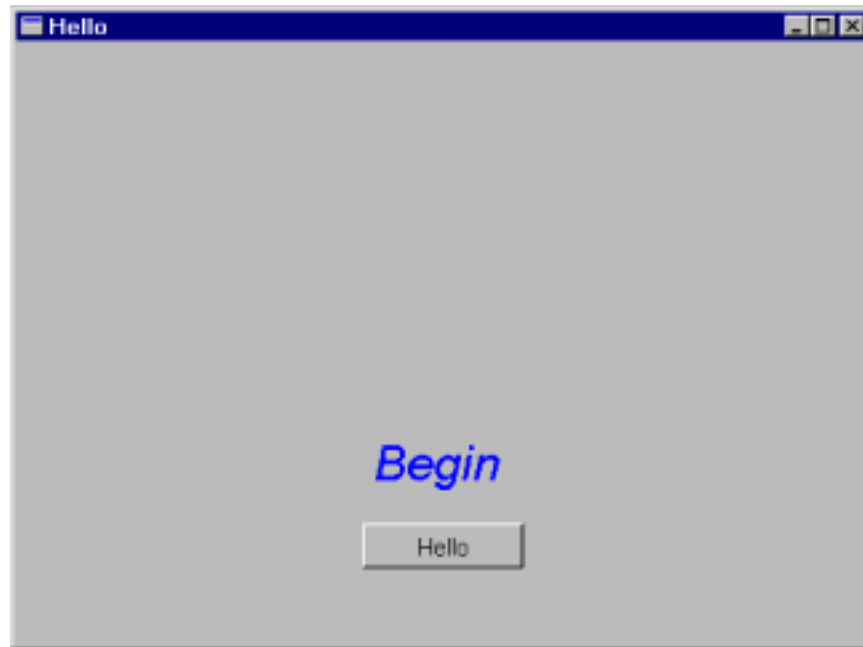



Figure 3.20. The rearranged Hello form

You are going to use the existing text control (CmStatic1) to display file names complete with drive and path. This means you need to change the font size back to a smaller point size. The reason for the smaller font size is to accommodate fully qualified file names that will be displayed in it.


Change the Font property for CmStatic1 to a size of 8 or 9 points. You also need to widen this text control to make room for long file names.

Click on the CmStatic1 control to select it and widen it to nearly the entire width of the form.

You are now going to add three new controls from the Toolbox palette.

Select the PowerCOBOL DriveList Control icon  from the Toolbox palette by clicking on it.

Move the mouse back to the Hello form and place the DriveList control near the top left corner of the form. Now right-click once on this new control to display a pop-up menu. Select Properties. Make sure the Auto Size option is checked and click on OK.

Select the PowerCOBOL FolderList Control icon  from the Toolbox palette by clicking on it.

Move the mouse back to the Hello form and place the FolderList control to the right of the DriveList control.

Select the PowerCOBOL FileList icon  from the Toolbox palette by clicking on it.

Move the mouse back to the Hello form and place the FileList control immediately under the FolderList control.

Now resize and move these three new controls until the Hello form looks something like the following:

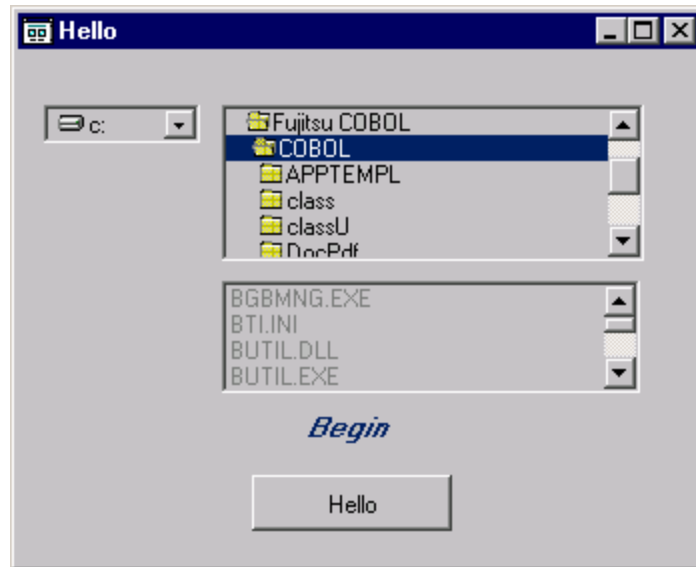


Figure 3.21. The Hello form with the three new controls added

You are now ready to begin writing the event procedure code to tie the behavior of these three new controls together.

Right-click once on the DriveList control to display a pop-up menu. Move the mouse to Edit Event Procedure and a secondary pop-up menu appears.

Move the mouse to the SelChangeEvent and left-click once on it. This displays the Editor for the event procedure for this event. This event will be triggered whenever the current drive selection is changed.

If a user selects a different drive, you need to ensure that the FolderList control display is updated with a list of folders on the newly selected drive.

To create this behavior, enter the following line of code in the PROCEDURE DIVISION:

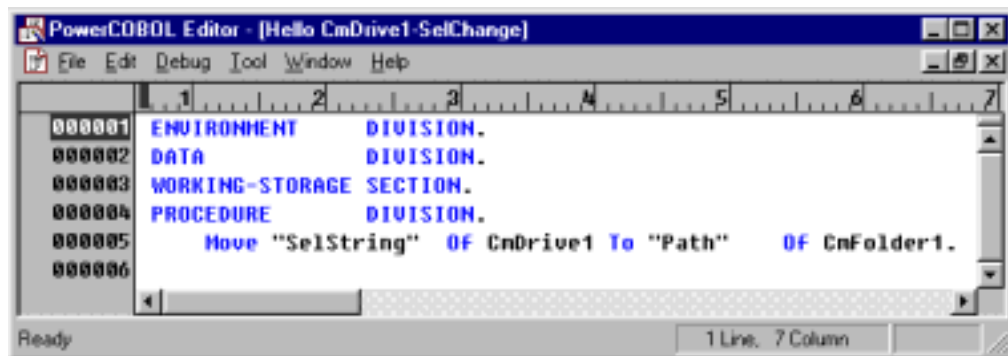


Figure 3.22. The event procedure code for the DriveList SelChangeEvent

Save this code and close the Editor. Note that whenever the PATH data field of the CmFolder1 control is updated with a new path name, it automatically retrieves a list of file names in the new path and displays them.

Any time a user clicks on a different drive in the DriveList control, the drive will be placed in the SELSTRING data item.

By moving SELSTRING of the DriveList to the PATH data field of the FolderList control, you ensure that the FolderList is automatically updated whenever a new drive is selected.

This single line of code placed in the appropriate location becomes a very powerful method to tie the behavior of these two separate controls together.

You now need to ensure that whenever a user clicks on a different folder in the FolderList control, the FileList control is updated with a new list of files.

Move the mouse over the FolderList control and right-click once on it to display a pop-up menu. Move the mouse over Edit Event Procedure and click on the Change event. Notice that we are going to use the Change event instead of the SelChange event so you can see the difference. By using the Change event, the list of files in the FileList control window will only change if the user double-clicks the mouse on a new folder to open it up. Simply selecting a new folder by using the up arrow keys or by left clicking only once on it will not open the folder – only double clicking on it will accomplish this and force the file list to be updated to reflect the contents of the new folder. If we used the SelChange event instead, the file list would be updated upon any new selection of a folder.

This should display the Editor with the event procedure for this event. Enter the following code:

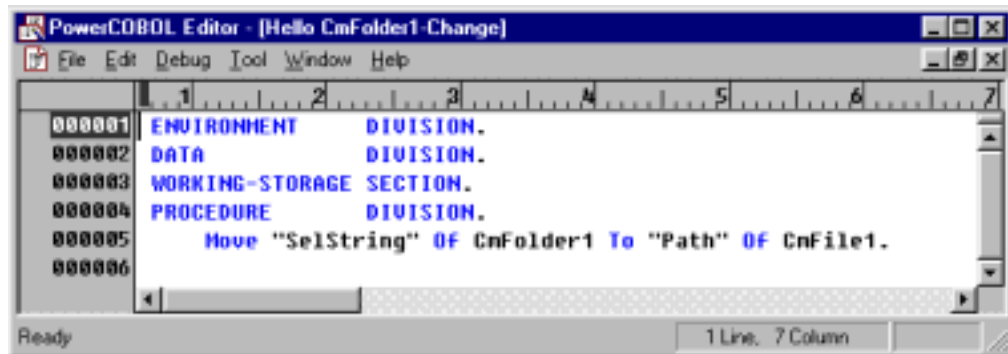


Figure 3.23. The event procedure code for the FolderList SelChange event

Once again, a single line of code becomes very powerful synchronizing the folder list control to the file list control.

Save this code and close the Editor. The FileList control behaves in a similar fashion to the FolderList control, in that whenever its PATH property is updated, it automatically retrieves a list of files in the new path and displays them.

This means that you need to ensure that whenever a user selects a different folder in the FolderList control, you should move the new path to the FileList control.

Because the FolderList control returns only the name of the current folder (and not its current path), you need to string together the current path along with a "\" and the new folder name to create the new path name.

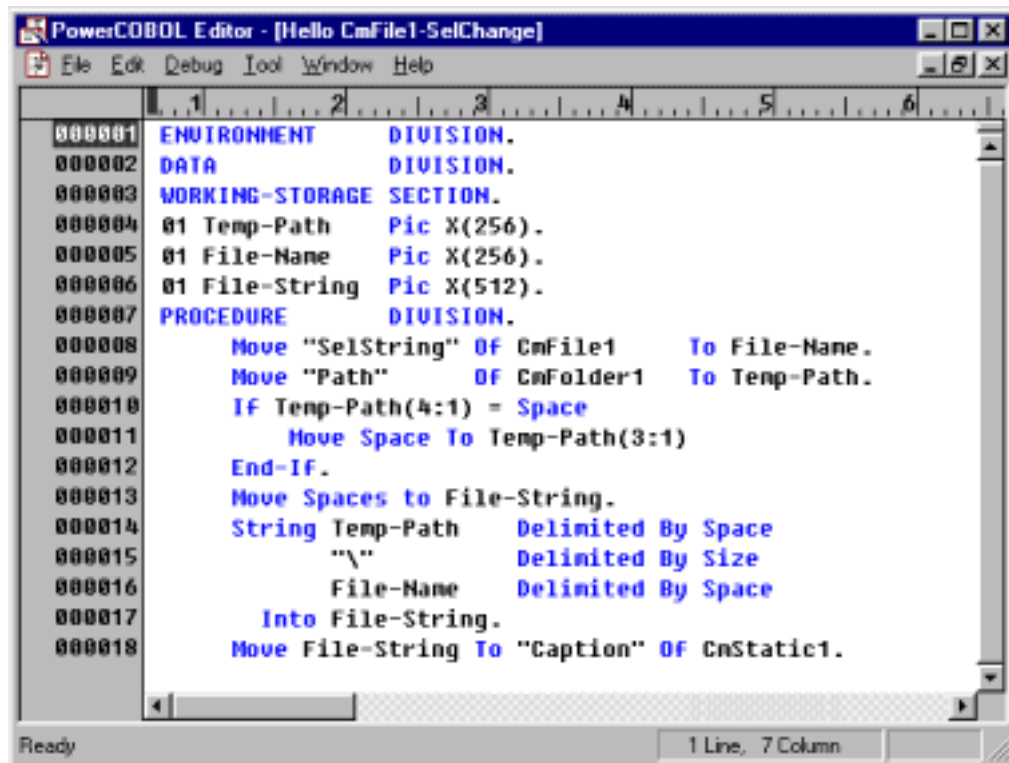
We need to check the path we receive to see if it already contains a "\" on the end to prevent adding another "\" and creating an error. The only case where this may

occur is when the path contains a root drive only (e.g. "C:\"). So we will add some logic to look for a space in the 4th character position in the path string which will signal us that we do indeed have a root drive only in the path and need to delete the "\" in character position 3.

Finally, you want to implement some behavior to ensure so that whenever a user selects a file name from the FileList control, the new fully qualified file name appears in the CmStatic1 text control.

Move the mouse over the FileList control and right-click once on it to display a pop-up menu. Now move the mouse to Edit Event Procedure and select the SelChange event.

This will display the Editor window for the SelChange event procedure. Enter the following code:



```

PowerCOBOL Editor - [Hello CmFile1-SelChange]
File Edit Debug Tool Window Help

000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Temp-Path Pic X(256).
000005 01 File-Name Pic X(256).
000006 01 File-String Pic X(512).
000007 PROCEDURE DIVISION.
000008     Move "SelString" OF CmFile1 To File-Name.
000009     Move "Path" OF CmFolder1 To Temp-Path.
000010     IF Temp-Path(4:1) = Space
000011         Move Space To Temp-Path(3:1)
000012     End-IF.
000013     Move Spaces to File-String.
000014     String Temp-Path Delimited By Space
000015         "\" Delimited By Size
000016         File-Name Delimited By Space
000017     Into File-String.
000018     Move File-String To "Caption" OF CmStatic1.
  
```

Figure 3.24. The event procedure code for the FileList SelChange event

Save this code and exit the Editor. In order to create the fully qualified file name selected, you are retrieving the current PATH from the FolderList control and stringing it together with a "\" and the actual file name selected from the FileList control.

You are then moving this fully qualified file name to the text area of the CmStatic1 control.

You are now finished with this round of enhancements. Close the Hello form edit session and go back to the Hello application in the Project Manager window.

Save the project and perform a Rebuild All on it to create a new executable. Correct any errors as needed until you have a clean build.

Now execute the application. You should be able to move around the disk drives and directories on your system. If you select a specific file, its fully qualified name should appear in the text box near the bottom of the screen.

When you are finished experimenting with the application, close it.

Adding Support to Print the Contents of the Window

You will now add a second push button to the form. This push button will be programmed to print the current contents of the form (window) when selected.

You will also see the powerful drag and drop editing integration between the PowerCOBOL tools. This feature allows you to drag controls over to COBOL scriptlet editing sessions to help you construct your code.

Make sure you have opened the Hello application for form editing.

Select the PowerCOBOL CommandButton Control Icon  from the Toolbox palette by clicking on it.

Move the mouse back over to the form and place the new push button to the right of the current Hello push button.

Edit the new push button's properties and change its caption to "Print".

The Hello form should now look something like the following:

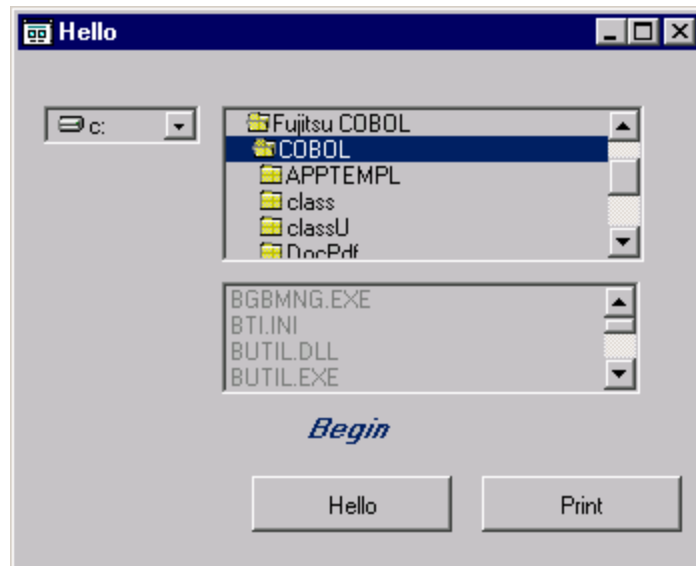


Figure 3.25. The Hello form with the Print push button added

Select the PowerCOBOL Print control icon  from the Toolbox palette to install support for printing.

Move the mouse back to the form and paste the Print control icon near the bottom left corner of the form.

This icon will not display at run-time. It is only present in the development window to remind you that you have elected to add printer support to the application. Therefore it does not actually matter where you paste it on the form.

You will now write the code to implement the printer support. The behavior you want is that any time the user clicks on the Print push button; the contents of the current form are printed.

It is a good idea to set a couple of the printer properties to ensure you have the correct paper type and printer orientation selected.

Move the mouse to the Print push button (not the Printer icon) and click once on it with the right mouse button to display a pop-up menu. Move the mouse to Edit Event Procedure and select the Click event from the secondary pop-up menu.

This will display the Editor on the Click event procedure code. You are going to drag and drop an icon from the Hello form into the Editor window.

In the Editor window, move the blinking cursor to column 12 of the blank line under the PROCEDURE DIVISION statement.

Make sure that you have repositioned the Editor window on your desktop so that you can see both the portion of the form being edited with the print control icon you just added and also the event procedure code in the Editor window.

Without closing the Editor window, move the mouse back over to the Hello form and over the print control icon you previously pasted there. Left-click once on this icon and hold the button down. Now attempt to drag this icon to the Editor window and drop it anywhere in the event procedure code.

This should cause the text "CmPrint1" to be inserted at the cursor position in the Editor, and to highlight it automatically. "CmPrint1" is the object name of the print process in this application.

Now right-click once on the highlighted string "CmPrint1" in the Editor window to display a pop-up menu.

Click on Insert Property near the bottom of the pop-up menu. This will display a secondary pop-up menu listing all of the available properties for this object.

Select the PaperType property. This will insert some additional code into the Editor at the appropriate location. The line you have begun to create should now look like:

```
"PaperType" OF CmPrint1
```

Now modify this line as follows so that it reads:

```
Move POW-PAPERLETTER TO "PaperType" OF CmPrint1.
```

This technique illustrates how to use drag and drop from the form being painted to the Editor to assist in creating supporting COBOL scriptlets. Note that "POW-PAPERLETTER" is a PowerCOBOL supplied constant that you can use instead of being forced to find the actual binary value required in Windows for this property. PowerCOBOL supplies a number of these constants to make programming easier. See the help system for further information on PowerCOBOL supplied constants.

You should also note from this example that you can highlight any PowerCOBOL application object name in the Editor and right-click once on it to display its associated methods and properties. You will find this useful in the future if you do not know specific properties or methods available for given objects.

Now go ahead and finish coding the event procedure code and comments as noted below. You may type these in manually or use the drag and drop feature and/or the pop-up menu options noted previously.

When you are finished, your event procedure code (scriptlet) should look something like the following:

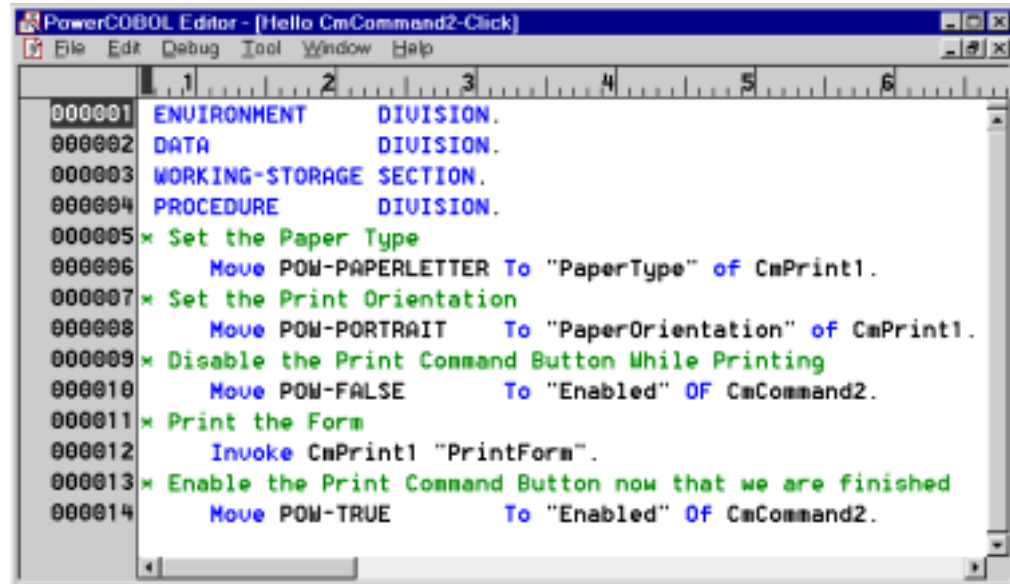


Figure 3.26. The event procedure for the Print push button event (printing the form)

Save the event procedure code and close the window by clicking on the close (X) button in the upper right corner. You may now save, rebuild and execute the application.

Using Supplied Methods and Properties

One of the most powerful features of PowerCOBOL is the use of the many control methods and properties provided. Each control comes with a set of predefined properties and methods.

You may easily make use of properties by moving various values to them, as you did above when changing the background color and font size of the static text control.

To obtain a list of the various properties available for a specific control on your form, you can right-click the mouse on the control name and select the Properties option from the pop-up menu. You can also look these up in the help system.

You may specify property values in this manner prior to application execution. If you want to change a control's property at run-time, as you did above, you need to code an appropriate COBOL Move statement to move a new value to the property.

Make use of the extensive on-line help system to look up property names to determine which values you may move into them, and what COBOL format these values must be in.

While in the Editor window, you may type in the name of a control anywhere in the edit session (or drag the control from the form or project manager into the Editor window to have its name dropped into the edit session).

Once you have the control's name in the Editor, highlight it and right click on it to bring up a pop-up menu. Select Properties to see a list of all of the properties

available. Selecting Methods will bring up a list of methods available for this control as well.

Use the on-line help system to answer any questions about available properties or methods. Using the Find tab in the on-line help system and typing in the name of the property or method will quickly take you to the relevant topics.

Using supplied methods will not only enhance your application, but will save you a great deal of development time.

In the printer example above, you can easily enhance the application by adding the SetPrinter method.

Adding a single line of event code such as follows:

```
INVOKE CmPrint1 "SetPrinter" RETURNING ReturnValue
```

will cause the following Print Setup dialog box to be displayed when executed:

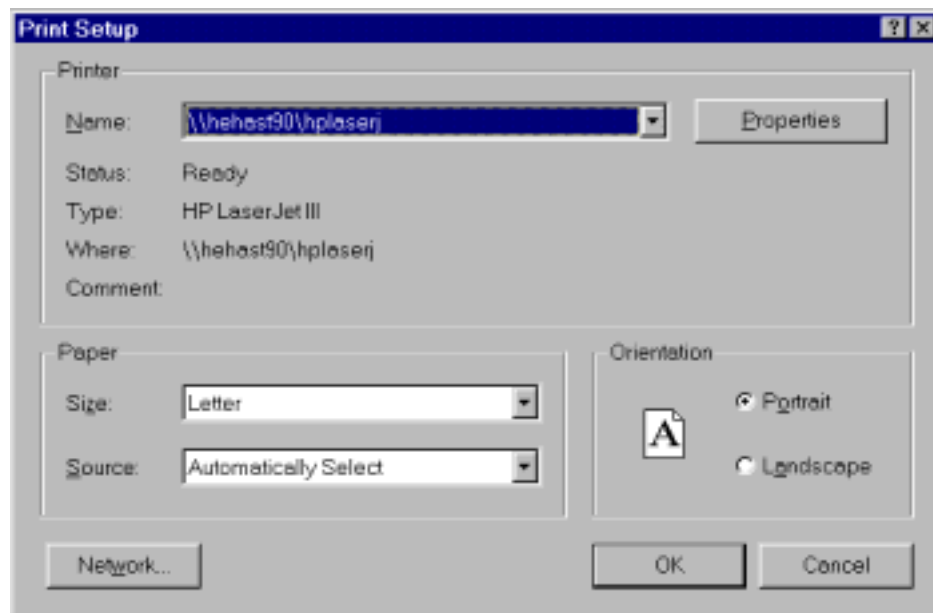


Figure 3.27. The Print Setup dialog box

This allows the user to select print properties and the actual printer he or she wishes to use for the print task.

Another print method available is SetPage, which will bring up the Page Setup dialog box as follows:



Figure 3.28. The Page Setup dialog box

Feel free to explore and experiment with the various supplied methods and properties.

When you are finished experimenting, close the application and go back to the Project Manager window. This completes this chapter.

Chapter 4. Creating and Editing an Application Window (Form)

PowerCOBOL makes it easy to develop an application with a highly customized graphical user interface. This chapter explains how to create and edit an application window (form) using PowerCOBOL. The following topics are discussed:

- Creating a form
- Setting the form properties
- Creating a control
- Setting control properties
- Manipulating controls
- Setting control order
- Placing controls in an array
- Setting colors
- Setting fonts
- Creating a menu bar
- Creating a toolbar
- Printing forms
- Printing procedure code

Overview

Using PowerCOBOL, you can drag and drop graphical controls onto a form (window) to generate the user interface of the application. In addition to creating the application windows, PowerCOBOL provides the following functions for:

- **Setting the form properties**
Select the size and properties of the form, the window border type, the title, and the existence of minimize/maximize buttons.
- **Creating controls**
Select graphical user interface (GUI) objects from the Toolbox palette and place them on the form.
- **Setting control properties**
Select the style and attributes (properties) of individual controls.
- **Manipulating controls**
Move, copy, resize, and edit controls or group controls together.
- **Setting control order**
Set control order to control how focus moves from control to control in an application window
- **Placing controls in an array**
Place controls in an array, as you would normally place variables in an array.
- **Setting colors**
Select colors to customize backgrounds, controls, text, etc.
- **Setting fonts**
Define the font, style, and size of the text in the interface.
- **Creating a menu bar**
Use the Menubar Editor to add menus to forms.
- **Creating a Toolbar**
Use the Toolbar control to add toolbars to forms.

Creating a Form

A new form named "MainForm" is automatically created whenever you create a new project in PowerCOBOL.

If you have not done so already, start PowerCOBOL to display the Project Manager window. Select New Project from the File menu and then select the "Standard Form" project template.

This will create a simple project with a single module (Main) and a single form (MainForm).

Now expand the project's objects by right clicking on the project name in the left windowpane and selecting Expand All from the pop-up menu.

There should be no controls associated with MainForm in this new project. If you see any additional controls associated with MainForm, then you may have selected the wrong template. Delete the current project and create a new one using the proper template.

Once you have defined a project, you may create additional forms for it from within the Project Manager window.

You can do this by right clicking on the name of the application module you wish to create the form under (for example, the "Main" module), and selecting Create Form from the pop-up menu that appears.

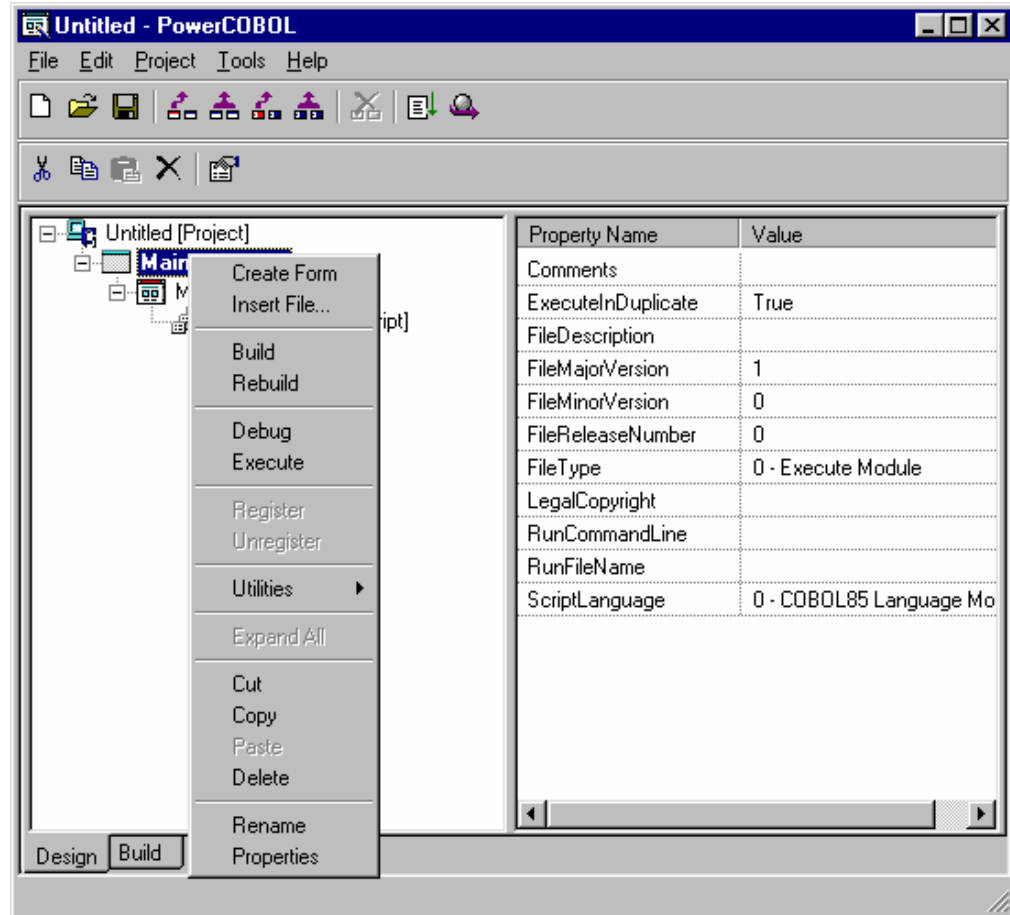


Figure 4.1. The Create Form option in the Project Manager pop-up menu

For now, do not create a second form, as you will be working on the default form (MainForm) you just created in the new project.

Once you have created a new form (either by creating a new project or using the Create Form option from an existing project), you may change its initial properties.

To change a MainForm's initial properties, left-click the mouse on MainForm's name in the left windowpane of the Project Manager window.

The form's properties and property values will then be displayed in the right windowpane.

Left-click once on the MainForm form to display its initial properties as follows:

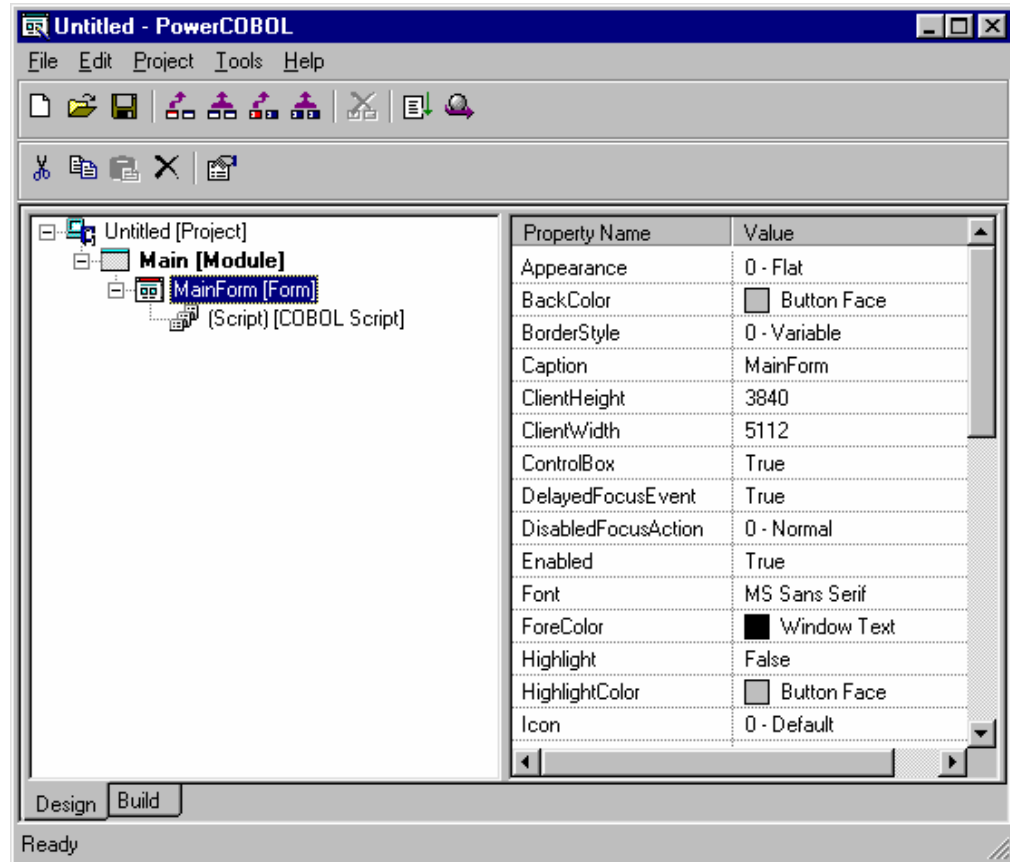


Figure 4.2. Displaying the initial properties of MainForm

Setting the Form Properties

This section describes how to develop the basic look and function of an application window. The process is called *setting the form properties*.

There are actually three different techniques that you can use to modify a form's properties. The first two techniques modify a form's initial properties when the form is displayed. The third technique allows you to actually modify a form's properties dynamically at run-time.

These three different techniques are as follows:

1. From within the Project Manager, you can select any form to display its current properties as shown in figure 4.2. You may then select any property in the right windowpane and modify it.
2. Once you actually open the form in the Form Editor, right-click the mouse anywhere on the actual form and select Properties from the pop-up menu. This displays the form's Properties dialog box as follows:

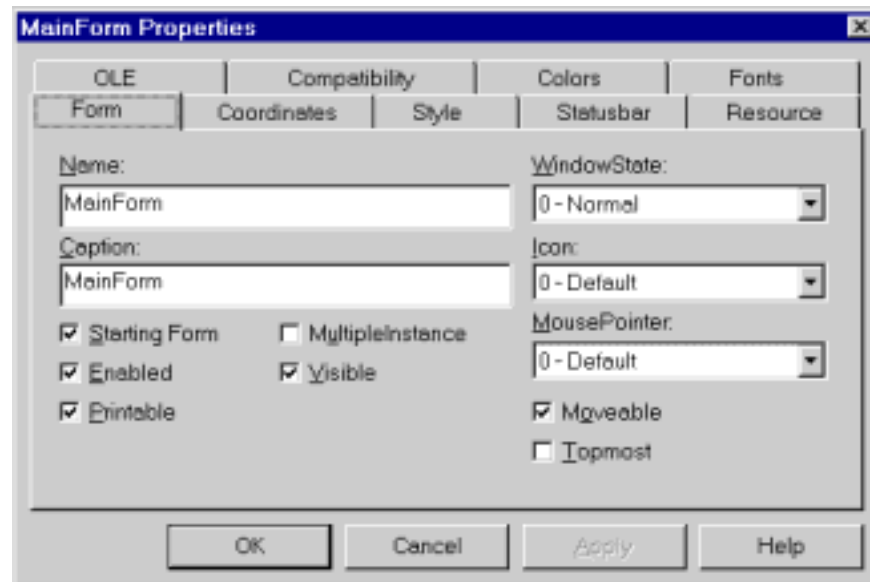


Figure 4.3. The Properties dialog box for a form

3. You may write COBOL scriptlets (event procedures) to modify a form's properties at run-time. This is discussed in Chapter 5.

Make sure you are in the Project Manager window and have selected MainForm to display its properties in the right windowpane as shown in figure 4.2.

You will now examine the actual properties to begin to understand the vast amount of graphical functionality available in PowerCOBOL.

Form Properties

Form properties may consist of a user-defined value, or a choice between True and False (on or off). They may also be selectable from a dropdown list containing many values.

To determine the values available for a specific form property, simply click in the property's value field. If a button appears in the far right side of the value field, click on it to view a dropdown list of available values.

The following properties are listed in alphabetical order under Property Name in the right windowpane of the Project Manager window:

Appearance

Defines whether the window (form) should appear as flat or with a 3D look when displayed on the screen.

BackColor

Defines the background color for the window.

BorderStyle

Defines the border style for the window.

Caption

Defines the title to be displayed in the title bar of the window.

ClientHeight

Specifies the current height of the window.

ClientWidth

Specifies the current width of the window.

ControlBox

Specifies whether the window will have a control box (the small boxes containing minimize, maximize and close buttons and options) when displayed.

DelayedFocusEvent

Specifies whether to delay the timing to generate the getting/losing focus events. This is used for compatibility with version 3 of PowerCOBOL.

Description (for ActiveX controls only)

Specifies the tooltip text for the ActiveX control.

DisabledFocusAction

Specifies the focus action when the Option button control is disabled. This is used for compatibility with version 3 of PowerCOBOL.

Enabled

Specifies whether the window shall be enabled (true) or disabled (false) initially.

Font

Displays the Font Properties dialog box. From here, you can select a font type, size and style.

ForeColor

Defines the foreground color of the window.

Highlight

Turns on (True) or off (False) highlighting.

HighlightColor

Defines the highlight color of the window.

Icon

Specifies an icon to be displayed when the window is minimized. A list of available icons will be displayed if you select this property.

IconName

Specifies the name of the icon associated with the window.

Left

Specifies how far from the left side of the screen the window should be displayed (typically used in conjunction with the "Top" property to select the initial position for the top left corner of the window on the screen).

MaxButton

Specifies whether the window should contain a maximize button.

MenuBarName

Specifies the name of the menubar used for the form.

MinButton

Specifies whether the window should contain a minimize button.

Mouse IconName

Specifies the name of the mouse icon.

MousePointer

Specifies which icon to use as a mouse pointer.

Movable

Specifies whether the window is movable by clicking on the title bar and attempting to drag it to a new location on the desktop.

MultipleInstance

Specifies whether to execute multiple instances of the form (window). Two or more instances of an application that include the same form may exist on a Windows system at the same time.

Name

Specifies the name of the Form.

Printable

Specifies whether the window shall be printable.

ProgID (for ActiveX controls only)

Specifies the Program ID that the ActiveX control will be registered under.

Scalable

Specifies whether to change the relative size, position and/or font size of controls on a form when the form is resized.

ScaleMode

Specifies the type of scale mode to use (pixels, mm, inches, and points) for the window.

ShowStatusBar

Specifies whether a status bar will be shown along the bottom of the window.

StartupPosition

Specifies the start up position of the window when it is first displayed on the screen.

StatusText

Specifies the initial text to be displayed in the status bar, if the show status bar property is turned on.

TitleBar

Specifies whether the window will have a title bar displayed.

ToolboxBitmap (for ActiveX controls only)

Allows you to specify an icon to be associated with an ActiveX control when it is added to a toolbox.

Top

Specifies the positioning of the window from the top of the screen when it is initially displayed. Typically used in conjunction with the Left property to define the position of the top left corner of the window on the screen.

Topmost

Specifies whether the form is taken to the top of the Z order (the order in which windows are layers on the desktop) and is kept there.

Visible

Specifies whether the window is visible on the screen when it is first initialized.

WindowState

Specifies whether the window's initial display state should be normal, minimized or maximized.

Creating a Control

This section describes how to create graphical user interface (GUI) controls on a form using either the mouse or the keyboard.

From the template application you created previously, move the mouse over MainForm[Form] in the left windowpane and right-click the mouse once.

This will display a pop-up menu. Select Open from the pop-up menu to open a form editing session on MainForm as follows:

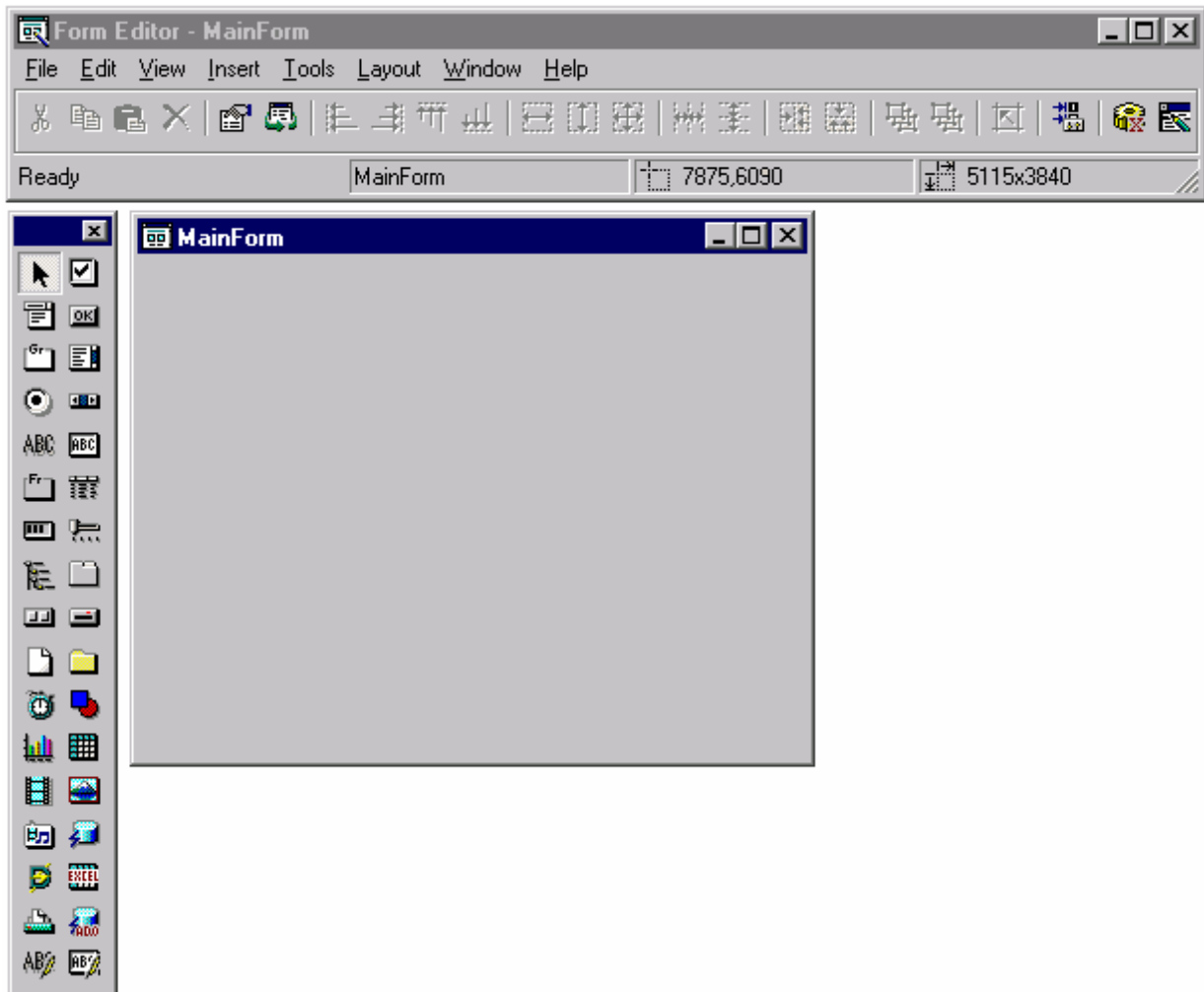


Figure 4.4. Editing a new form

Selecting a New Control

To create a new control from the available selections, you may select it from the Toolbox palette window or from the Insert menu in the Form Editor.

To select a control from the Toolbox palette, position the cursor over the control to be created in the Toolbox palette and click on it with the left mouse button. The control button will appear 'pushed in' to indicate that it has been selected.

You may then move the mouse back over the form to where you wish to place the newly selected control and the control will appear on the form.

To select a control from the Form Editor menu, select Insert Control from the Insert menu. The following menu will appear for you to select an available control:

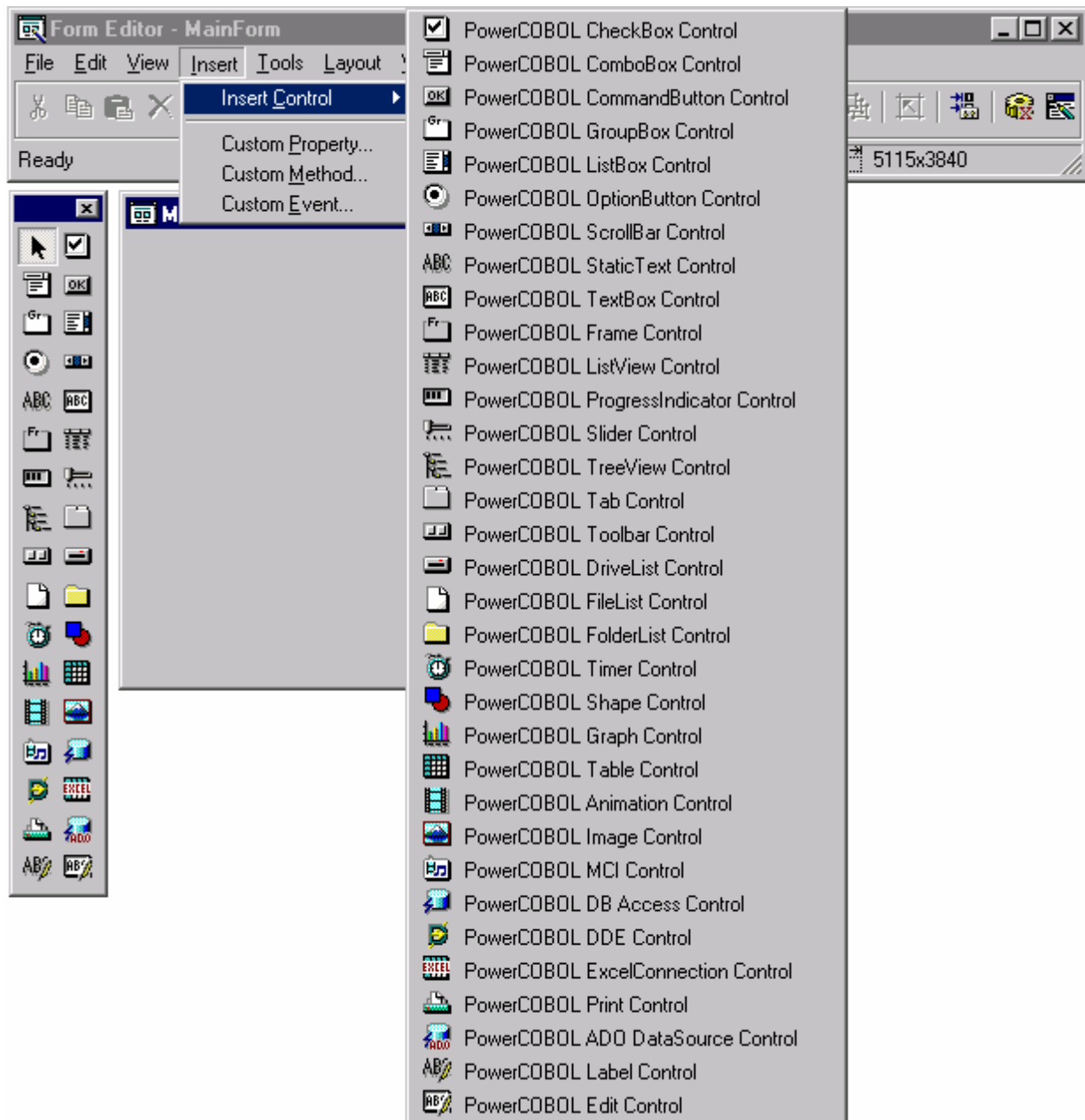


Figure 4.5. The Insert Control menu

Move the mouse to the control you desire and left-click once to select it.

After you have selected a control (from the Toolbox palette or the Insert Control menu), move the mouse back over to the form and move the control to the appropriate location. Left-click the mouse a second time to paste it on the form.

Sizing and Positioning a Control Using the Mouse


To resize a control you must first select it with the mouse. When a control is selected, it will have a border with a series of square dots around it.

To resize a control that is currently selected, press and hold down the left mouse button on one of the selection borders and drag it to change the size of the control you are creating.

When the control is the desired size and in the position that you want, release the mouse button.

To reposition a control, move the mouse inside the selection border so that it is not touching any part of the selection border. Then left-click the mouse and while holding the mouse button down, move the control to the desired location and release the mouse button.

Canceling the Action in Progress

Move the mouse to the Ready icon  in the top left of the Toolbox palette. Left-click once on it to stop the creation of a control after you have selected it but have not already pasted the control down on the form.

If you have already pasted the new control down on the form, and wish to undo this action, make sure the new control is currently selected. You may then delete it by right clicking the mouse on it and selecting Delete from the pop-up menu, or by selecting Delete from the Edit menu.

Setting a Control's Initial Properties

This section describes how to develop the basic look and function of the individual controls in an application window. This process is accomplished by manipulating the control's initial properties.

There are two different processes available to manipulate a control's properties. The first technique is to access the control's properties from within the Project Manager window by selecting the specific control in the left windowpane and accessing its available properties in the right windowpane.

Once you are in the Form Editor, you may still move to the Project Manager window and manipulate form and control properties without closing the form editing session.

If you change a form's properties or one of its controls' properties in the Project Manager window, the change will appear immediately in the active form editing session.

There is a second technique for manipulating form and controls properties from within an active form editing session, without returning to the Project Manager window.

To do this, simply right-click the mouse once on the control or form whose properties you wish to manipulate and select Properties from the pop-up menu.

Alternatively, after selecting a form or control whose properties you wish to manipulate, you may select Properties from the Edit menu.

For example, if you move the mouse over a push button control named CmCommand1 and right-click the mouse once on it and then select Properties from the pop-up menu, you will be presented with the following window:

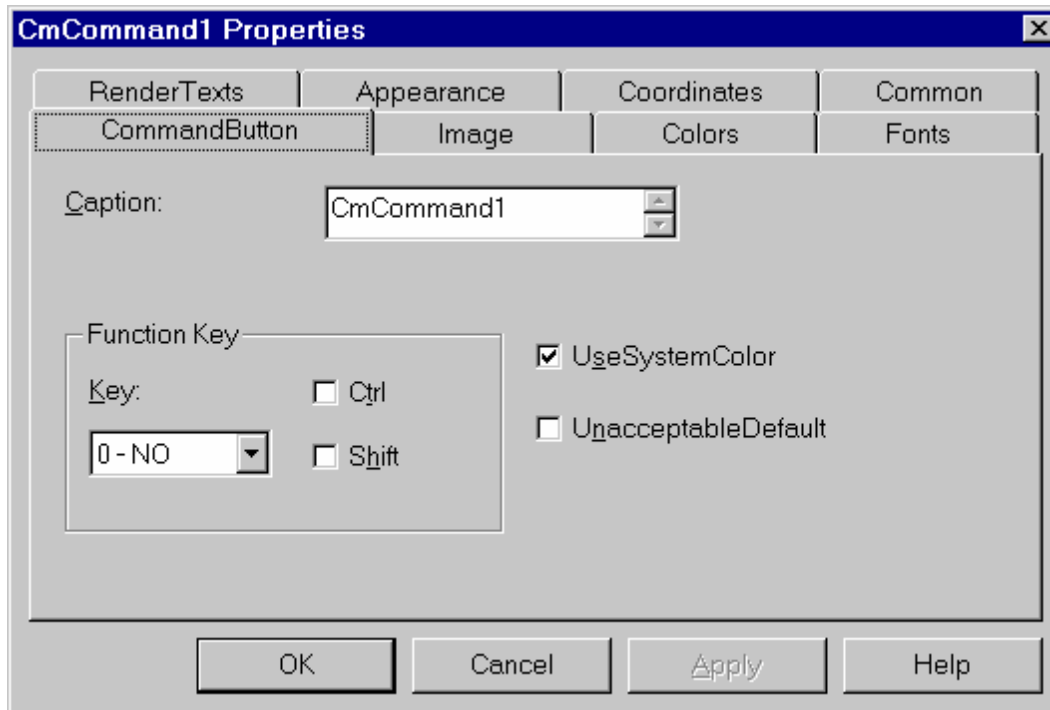


Figure 4.6. The Properties dialog box for a command button control

You may access this control's properties by selecting the appropriate tab in the Properties dialog box and changing any specific property. Clicking on Apply or OK after changing a property will apply the change immediately.

Manipulating Controls

Once you have created controls on a form, you can easily move, copy, resize, or edit them. You can also group several controls together to move and manipulate them as a group.

Copying a Control

To create a control with the same properties as that of another control, simply copy it by using the Copy and Paste options in the pop-up menu. You may also use the Copy and Paste commands in the Edit menu. When you do this, the new control created when you paste it in is given a unique name following the PowerCOBOL default naming convention.

Follow these steps to use the Copy and Paste commands in the pop-up or Edit menus.

1. Select the control to be copied by clicking on it with the left mouse button.
2. Select Copy from the Edit menu or right-click once on the control and select Copy from the pop-up menu.

3. Move the mouse over the form but not over any control and right-click once to display a pop-up menu.
4. Select Paste and a copy of the control will be placed in the upper left-hand corner of the form.
5. Move the mouse to the newly created control and right-click once on it. Holding the mouse button down, drag the control to the desired location and release the mouse button to paste it down on the form.

Note: To cancel the Copy operation, select the new control and select Delete from the Edit menu or right-click once on the new control and select Delete from the pop-up menu.

Editing a Control

Controls can be edited from the Edit menu. The following are the standard editing commands in Windows environments:

Cut

Removes the selected control or text from the editing area and stores it on the clipboard. At the next Cut or Copy command, the selection is erased from the clipboard.

Copy

Copies the selected control or text to the clipboard without removing the selection from the editing area.

Paste

Places the clipboard contents into the editing area at the location of the cursor.

Note: The Paste command usually follows the Copy or Cut command.

Delete

Removes the selected control or text from the editing area.

Select All

Selects all controls on a form. This is useful if you wish to perform the same operation on multiple controls.

Properties

Displays the Properties dialog box for the currently selected control or form.

The editing functions can also be issued using keyboard key combinations. Use these keys or key combinations to accomplish the same editing functions that are available in the Edit menu.

CTRL + X	(Cut) Removes the selected control or text from the editing area and stores it on the clipboard. At the next Cut or Copy command, the selection is erased.
CTRL + C	(Copy) Copies the selected control or text to the clipboard without removing the selection from the editing area.
CTRL + V	(Paste) Places the clipboard contents into the editing area at the location of the cursor.
DELETE	Removes the selected control or text from the editing area.
CTRL + A	(Select All) Selects all controls on a form. This is useful if you wish to perform the same operation on multiple controls.

In *insert* mode, newly entered characters push the string ahead of the cursor. In *overwrite* mode, newly entered characters delete and overwrite existing text. The Paste command works in either insert or overwrite mode.

Batch Editing Controls

Batch editing allows you to edit several controls at the same time. This function allows you to perform standard editing commands on a group of controls in the same manner that you do with individual controls. Batch editing supports the Copy, Cut, Paste, Delete, and Move commands as well as color and font setting functions.

Canceling Batch Editing

White control handles surround grouped controls. A blue border surrounds the currently selected control within a group. Cancel batch editing either by removing individual controls from the group, or by deleting the entire group selection.

To remove a control from a group, click on a grouped control and press the SHIFT key.

To cancel the batch editing process at any time, left-click the mouse once on a control or area of the form that is not part of the group.

Grouping Controls

Grouping relates controls, such as radio buttons, to one another.

Controls can be grouped using the mouse.

Follow these steps to group controls using the mouse.

1. Move the cursor to a clear part of the form, ideally adjacent to the controls to be grouped.
2. Press and hold down the left mouse button and drag the cursor across all of the controls to be grouped. This action draws a frame around the controls.
3. When all of the controls to be included in the group are framed, release the left mouse button.

Alternatively, you can create a group by pressing the SHIFT key as you select individual controls. Follow these steps to group controls by clicking on them.

1. Select the first control to be included in the group by clicking on it with the left mouse button.
2. Press and hold the SHIFT key and simultaneously select the other controls to be included in the group. White control handles display around the controls that are included in the group. A blue border is displayed around the currently selected control within a group.
3. When all of the controls to be grouped have been selected, release the SHIFT key.

Aligning and Sizing Groups of Controls

When you create a new project and open a new form in form editing mode, the form's background does not contain a grid.

You can, however, view a grid during form editing to assist in aligning controls. You can turn on the grid by selecting Grid from the Tools menu. A dialog box appears and allows you to enter a grid size and to turn on display of the grid.

A value of 10 points in both the Width and Height fields is typically a good choice. The dialog box should thus look something like the following:

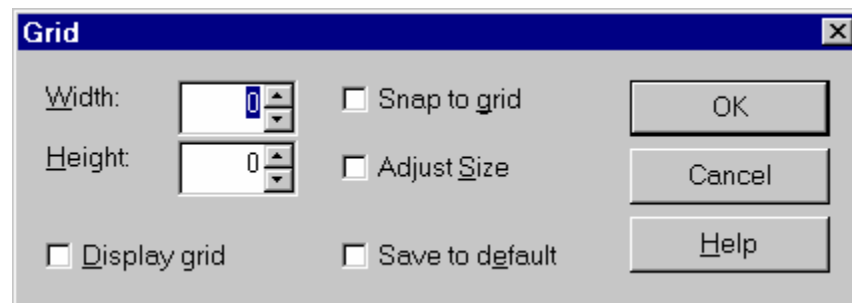



Figure 4.7. The Grid options dialog box


When the Display Grid feature is checked, it is easier to manually align controls to the grid. When using the automatic alignment tools, it does not matter whether the grid is turned on or not.


To do this, click the Align to Grid Icon , or simply right-click once on any control and select Align to Grid from the pop-up menu.

Additionally, the PowerCOBOL Form Editor provides several powerful functions for aligning controls. Several of these functions are provided to work on groups of controls.


These functions are available from the Layout Menu or as icons directly on the Form Editor tool bar.


These include the following functions that work on single or groups of controls:


 Right and Left Centering - Centers a control or group of controls between the right and left borders of a form.


 Bottom and Top Centering - Centers a control or group of controls between the top and bottom borders of a form.


Additionally, the following functions work only on groups of controls. Note that alignment is always relative to the one selected control in the group (surrounded by a blue border). White borders will surround the other controls in a selected group. You may change the relative control (blue bordered control) by simply left clicking on another control in the group).


 Adjust Left - Aligns the entire group along a straight left edge.


 Adjust Right - Aligns the entire group along a straight right edge.


 Adjust Top - Aligns the entire group along a straight edge that is parallel with the top border.


 Adjust Bottom - Aligns the entire group along a straight edge that is parallel with the bottom border.

 Width Match - Resizes all controls in the group so that their width matches the control in the blue border.

 Height Match - Resizes all controls in the group so that their height matches the height of the control in the blue border.

 Size Match - Resizes all controls in the group so that their sizes match the width and height of the control in the blue border.

 Horizontal Space Equalization - Arranges the controls in the group so that they are evenly spaced on a horizontal line.

 Vertical Space Equalization - Arranges the controls in the group so that they are evenly spaced on a vertical line.

Setting Control Order

The way controls are *ordered* controls how they are displayed and in what order the cursor moves through the fields.

In Windows applications, the TAB key and SHIFT + TAB keys move the cursor forward and backward respectively through the fields on the form.

In PowerCOBOL, the order in which controls are *created* on a form will control the default order in which the cursor moves from control to control.

Note that this means that the physical order in which the controls actually appear may not necessarily dictate the default order in which the cursor flows between them.

Changing the Tab Order of a Control

If you wish to change the tab order of a control, select that control individually by right clicking the mouse on it to display its Properties dialog box. Click on the Common tab, and the dialog box should look some like the following (note that the following dialog box is for a push button control - some property dialog boxes vary depending on the type of control selected):

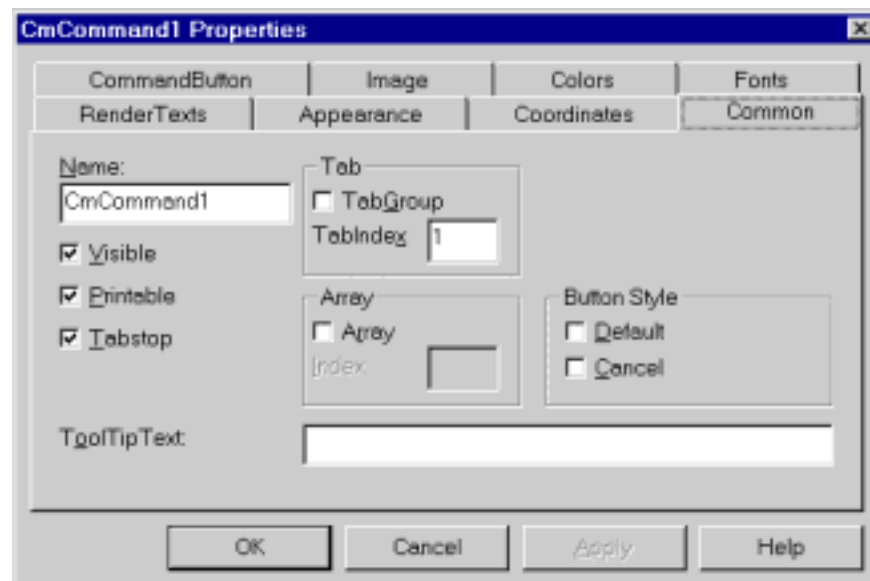


Figure 4.8. The Common Properties dialog for a push button control

The number in the TabIndex field represents the numeric order in which each control will be tabbed to. In the above example, the CmCommand1 push button is currently the 1st control in the cursor movement order.

If you modify this value, other Controls will automatically have their tab index numbers modified accordingly.

The default number in this field will be the actual order in which the control first appeared on the form. For example, if you place a push button control on the form, and it is the third control you have created, its tab index number will be set to 3.

This will occur regardless of its physical position on the form, and it will be the third control in the order in which the cursor moves from control to control.

Beginning with version 5.0, PowerCOBOL now provides a very useful tab ordering utility. You can invoke it by selecting Tab Order from the Layout menu, or by clicking

on the Tab Order icon  on the Form Editor's toolbar.

Setting Control Tab order using Tab Order Utility

In Windows applications, the TAB key and Shift + Tab keys move the focus forward and backward respectively through the fields on the form. In PowerCOBOL, you can set the order in the Tab Order utility window easier than you set it in the properties dialog.

To change the tab order of a specific control, simply hold the left mouse button down on it's icon in the left side of the Tab order window and drag it up or down to the desired tab location and let up on the mouse button.

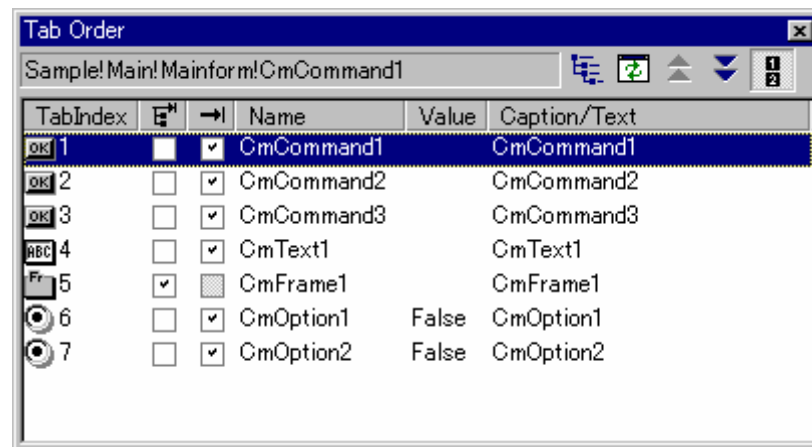


Figure 4.9. The Tab Order dialog box.



Group view switch

Switches the indication between index order and group order.

Note that the controls which do not have a TabIndex property are not listed in the window. (e.g. Image control, Timer control, etc.)

When the button isn't pushed, the Tab Order window indicates the index order of the controls as follows:

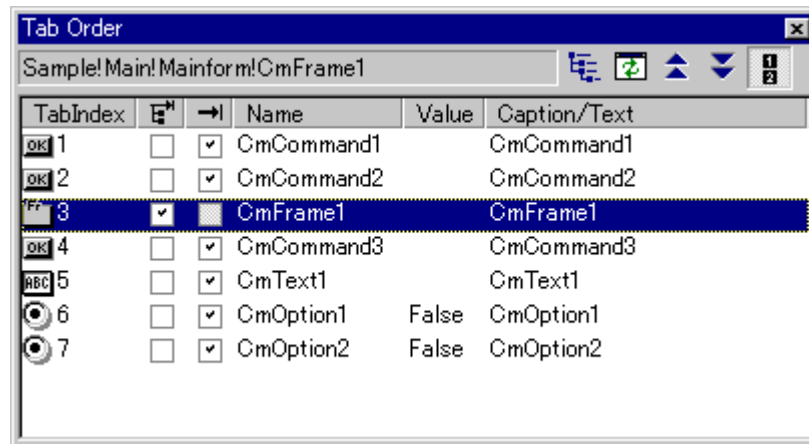


Figure 4.10. The Tab Order dialog box - Index Order.

When the button is pushed, the window indicates the group order with a tree style:

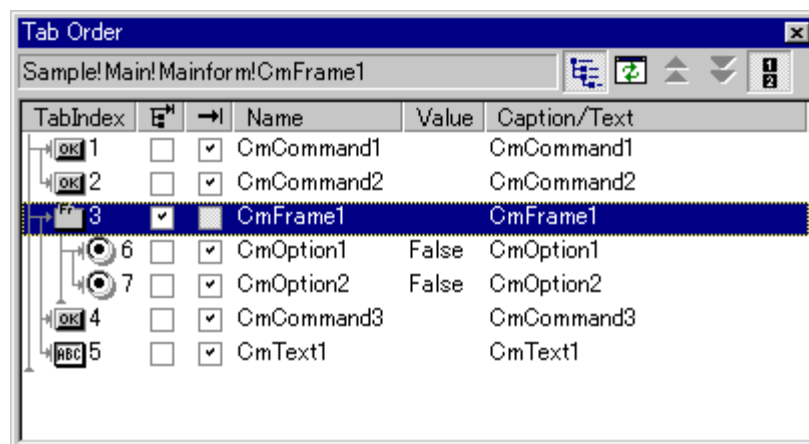


Figure 4.10. The Tab Order dialog box - Group Order.



Refresh the list

If you click this button the list refreshes.



TabIndex setting

When the window indicates the index order, you can change the order of the TabIndex property using these buttons, and you can change the index with Drag&Drop operation.



Display TabIndex label

When you are editing the form window, if you click this button you can see the TabIndex number in the form window.



TabGroup setting

You can set the TabGroup property in this column.



TabStop Setting

You can set the TabStop property in this column.

Placing Controls in an Array

Controls on a form can be placed in an array. Each control is then identified with an index value that is automatically determined by PowerCOBOL.

An array may consist only of the same type of controls. You may not mix different control types within a single array. Controls of the same type will be assigned the same name (they are accessed individually with a subscript) and will share the same event procedures. They do not have to share the same properties, however.

If you group together multiple types of controls (for example, you include 3 push buttons and 2 check box controls in a single group), and then right-click once on one of these controls to create a new array, only one array will be created. The array that is created in this case will only include the same control types along with the actual control you actually clicked on to create the new array.

If you created the previously noted group, and right-clicked on one of the command buttons to create a new array, an array of push button controls would be created with the three push buttons. The two check boxes included in the group would be completely ignored. They would not be included in the push button array, nor would a separate array be created for them.

1. To create an array, click on the first control in the form to be selected for the array.
2. Press and hold the SHIFT key and click on each subsequent control to be included. Alternatively, you can press and hold the left mouse button and drag it across the controls to select them (a black border temporarily appears to outline the controls you are selecting).
3. Click on the right mouse button to access the pop-up menu. Move the mouse to Array. This will display a secondary pop-up menu. Click on New Array.

An informational dialog box appears for you to display the name of the newly created array and the number of additional controls added to it. Click on the OK button to close this dialog box.

Note: When controls are selected for an array, they display blue or white 'handles' (the first control has blue handles and subsequent controls have white ones) on each side.

Creating an Array from a Group of Controls

Grouped controls can be placed in an array in the same way that individual controls can. Select the group of controls. Click on the right mouse button to access the pop-up menu. Choose Array, and then select New Array.

Controls are placed in an array in the order in which they were grouped. Other controls added to the array do not become part of the grouped control, even if they are the same type of control.

Adding Controls to an Existing Array

You can add controls to an existing array in the following manner:

1. Use the mouse to select all of the controls you wish to add to an existing array.
2. Right-click the mouse once on one of these newly selected controls to display a pop-up menu.
3. Move the mouse to Array to display a secondary pop-up menu. Instead of selecting New Array, select the name of the existing array that you wish to add the currently selected controls to.

An informational dialog box appears to confirm that the new controls were added to the existing array. Click on the OK button to close this box.

When a new control is added to an existing array, it is added to the end of the array and PowerCOBOL automatically assigns its index value.

Note: You cannot add a control to an array using the *cut* and *paste* clipboard functions.

Copying Controls in an Array

Copy an arrayed control the same way you would copy a control that is not in an array. When an arrayed control is copied, it is not automatically added to the end of an array. You must add it to the array using the method described previously.

Deleting Controls in an Array

Delete an arrayed control the same way you would delete a control that is not in an array. When an arrayed control is deleted, the index values of all the other controls in the array are adjusted as necessary.

Removing Controls from an Array

To remove controls from an array, right-click once on the control, move the mouse to Array in the pop-up menu, and select Remove From Array.

The released control is moved outside the array and the index values of the remaining controls are adjusted as necessary.

Grouping Controls Using a Group Box or Frame Control

PowerCOBOL provides two separate controls for grouping controls together logically on a Form. If you have a series of check boxes or Radio buttons and you wish to group them within a sub area on a form, you can use either of these controls to accomplish such. See the PowerCOBOL help system for more information on using these grouping controls.

Setting Colors

To change the background or foreground color of a form or control, right-click once on it and select Properties from the pop-up menu. Then select the Colors tab from the Properties dialog box that appears as follows:

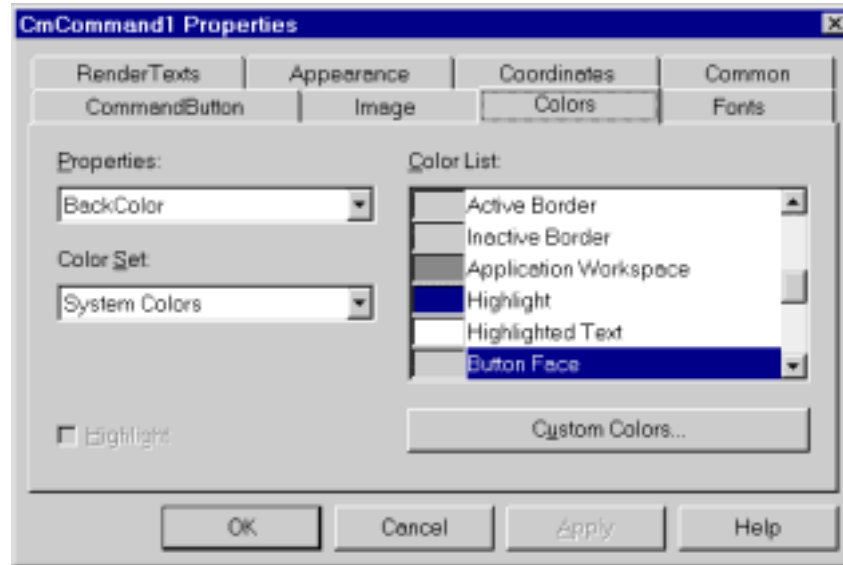


Figure 4.11. The Colors tab of the CmCommand control Properties dialog box

This dialog box allows you to alter the background, foreground and highlight color of the associated control.

There are three color sets provided from which to choose colors:

System Colors

These are colors defined by the operating system for specific control behavior.

Standard Colors

These are the standard colors defined by the PowerCOBOL.

Custom Colors

This displays a color palette as follows:



Figure 4.12. The custom Color palette

You may select a basic color from here, or if you desire something a bit more customized, click on the Define Custom Colors button and you will be presented with the Define Custom Color dialog box.

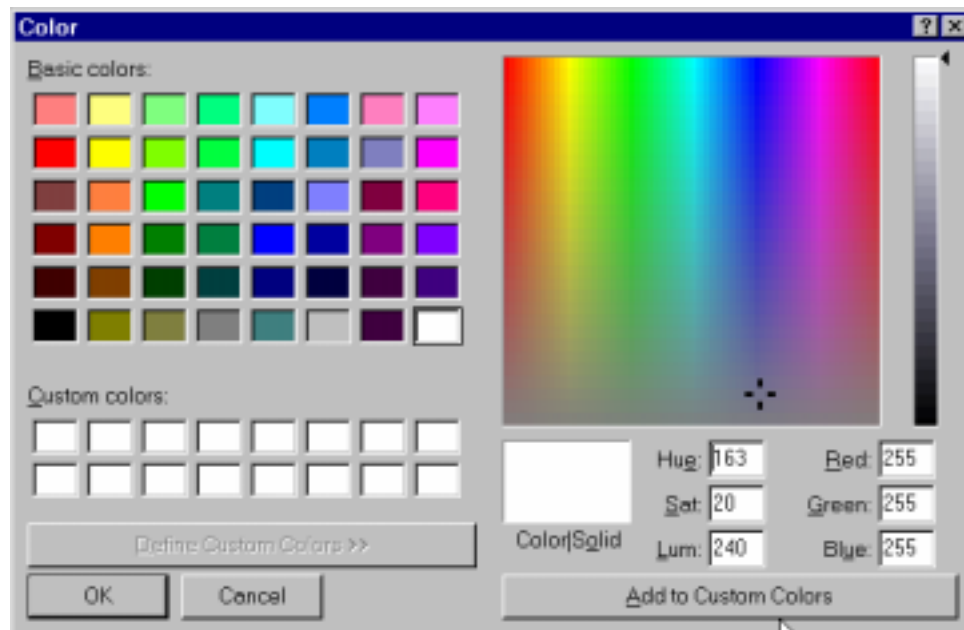


Figure 4.13. The Define Custom Color dialog box

Using this dialog box, you may move the cross hair icon in the color window on the right to select a specific color range.

Then move the small arrow tip at the top right of the vertical color bar to select the specific color from the range represented.

As you do this, the actual color selection will appear in the larger white box entitled Color|Solid.

Alternatively, you may manually type in values for Hue, Saturation, Luminance, or RGB to create a color.

Once you have selected a color, you may click on the Add to Custom Colors button and your new color will be added to one of the white boxes in the Custom Colors field.

You may then select your custom color from the palette and apply it.

Setting Fonts

This section explains how to use the Fonts tab to select and change fonts. You can set and change the type, size, and style of text in controls.

To open the Fonts tab in the Properties dialog box, right-click once on a form or control and select Properties. When the Properties dialog box appears, click on the Fonts tab.

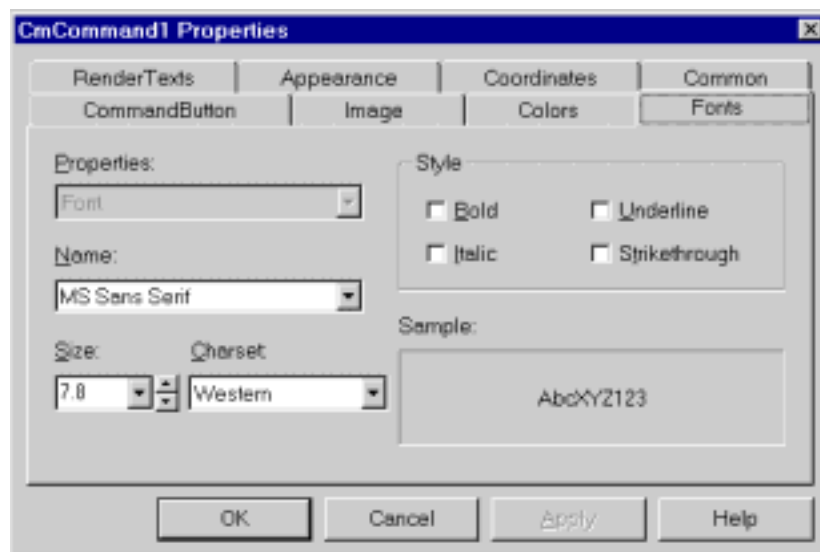


Figure 4.14. The Fonts tab in the Properties dialog box

From this tab, you may select a new font, font size, and/or style. As you alter the selection, a sample of the newly selected Font property will be displayed in the Sample text box.

To select or change a font:

To select or change a font, click on the down arrow button to the right of the Font field. A dropdown list of available fonts is displayed. Click on the name of the font you want to use.

To change the font size:

You can change the font size by clicking on the down arrow button immediately to the right of the Size field. A dropdown list of pre-set sizes is displayed. Click on the desired font size or click on the Size field and type in a specific font size. Another way to change the font size is to use the up and down arrow buttons to the right of the Size field down arrow. Click on the up arrow to increase the point size. Click on the down arrow to decrease the point size.

To change the style of a font:

Four styles are displayed in the Style box: bold, italic, underline, and strike. Click on the box to the left of each style to apply it to the selected text control.

Creating a Menu Bar

With PowerCOBOL you can create menu bars and pop-up menus using the Menu Editor. The following topics describe how to use the Menu Editor.

Creating and Editing a Menu

You can access the Menu Editor by either invoking the Menu Editor dialog from the Tools menu, or by using the Menu Editor sub-menu on the form's pop-up menu. The Menu Editor dialog gives the ability to delete menus as well as select or create them, whereas you cannot delete menus using the pop-up menu.

The Menu Editor Dialog

You display the Menu Editor dialog by selecting "Menu Editor" from the Tools menu.

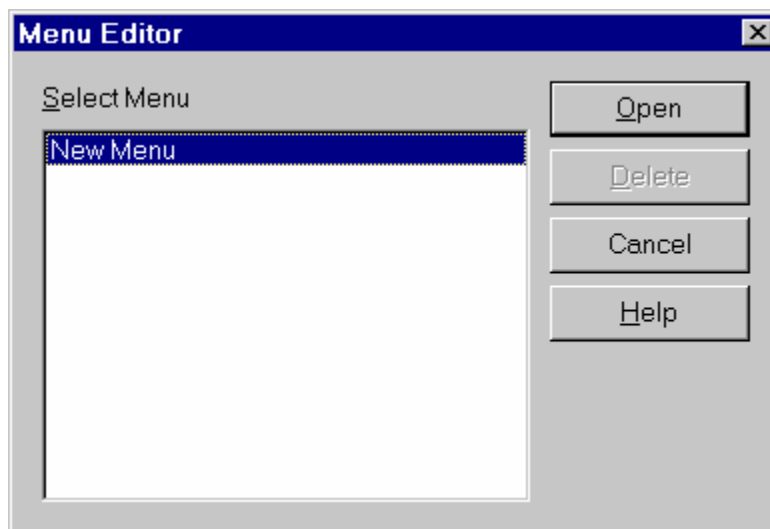


Figure 4.15. The Menu Editor

The parts of the Menu Editor dialog are:

Select Menu List Box

This list box contains the menus defined for this form.

Either select the menu you wish to edit or delete, or select "New Menu" to create a new menu.

Open Button

Click on the Open button to edit the selected menu or create a new menu.

Delete Button

Click on the Delete button to delete the selected menu. If you are editing that menu, the Form Editor closes the Menu Editor automatically.

Using the Pop-up Menu

To access the Menu Editor from the form's pop-up menu, right click on the form to display the pop-up menu. Move the mouse over the "Menu Editor" function and, from the sub-menu, either select the name of the menu you wish to edit or select "Create Menu" to create a new menu.

Associating a Menu with a Form

PowerCOBOL assumes that the first menu you create is the menu bar for the form.

To change this association, display the form's Properties dialog (e.g. by double clicking on the form) and select the Style tab. You can then select a different menu, or "None" in the MenuBarName drop-down list.

Using the Menu Editor

When you open a menu the Menu Editor window is displayed and the Form Editor Edit menu and toolbar change to provide Menu Editor functions. The Menu Editor functions are also available on pop-up menus:

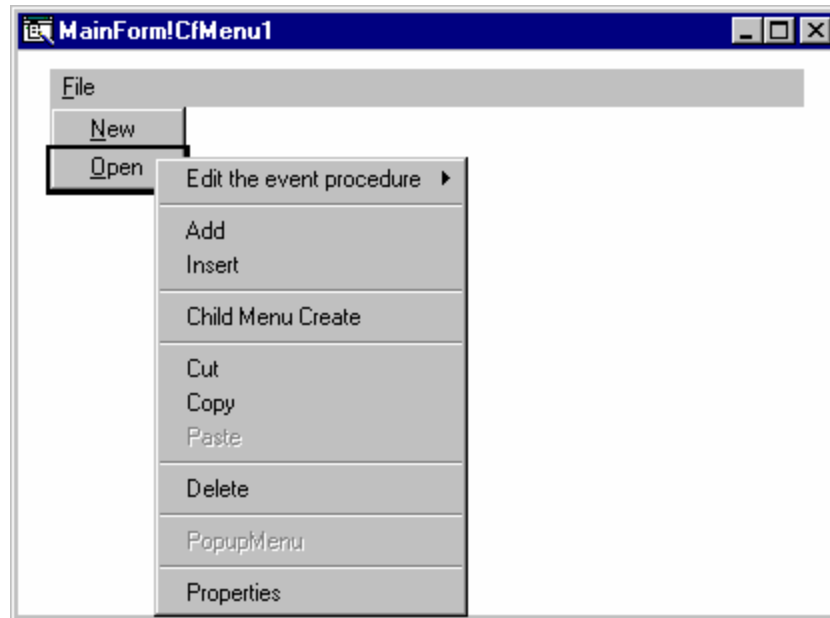


Figure 4.16. Adding options to a menu.

To create a menu item, right click on the menu bar, or a menu item, and select Add or Insert from the pop-up menu.

To create a child menu, select "Child Menu Create" from the pop-up menu.

The Menu Editor inserts blank menu items when you use the Add, Insert, or "Child Menu Create" functions. You then define the menu item by editing its properties.

Editing Menu Item Properties

When you have created a menu item you edit its properties in the Menu Properties dialog. To display the Menu Properties dialog either double click the menu item, or right-click the menu item and select Properties from the pop-up menu.

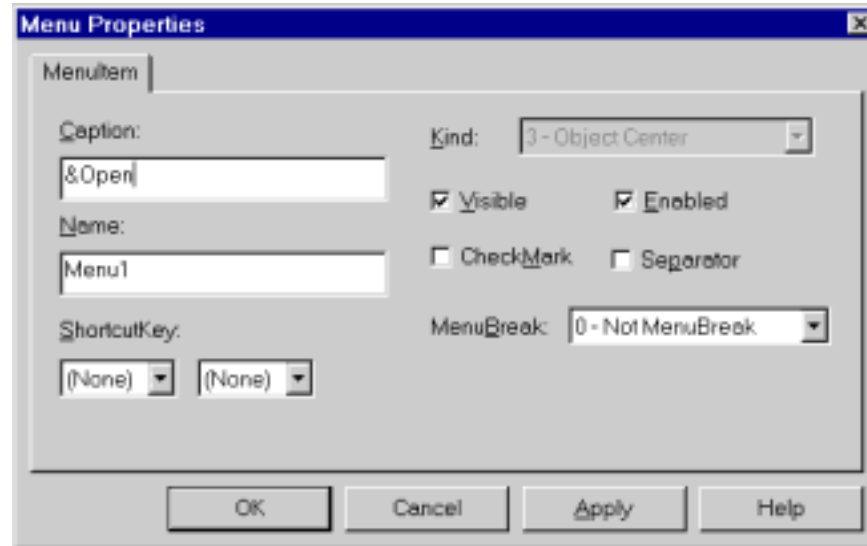


Figure 4.17. Menu properties.

The MenuItem properties are:

Caption:

Specifies the character string to display in the menu item.

If you want to set an access key precede the character with "&". For example, to make "O" the access key for the Open menu function you would enter "&Open" in the Caption field.

Name:

Contains the name of the menu item.

ShortcutKey:

Sets the key to use as a shortcut key.

Select None if you do not want to assign a shortcut key.

Note that you cannot assign shortcut keys to the top-level menu items nor to popup menu items.

The shortcut keys you can specify are:

[F1] to [F24],
[Del],
[Ctrl+A] to [Ctrl+Z],
[Ctrl+F1] to [Ctrl+F24],
[Shift+F1] to [Shift+F24].

NOTE:

1. The keys you can use as function keys (F1 to F24) depend on the Windows system and model. Therefore confirm if you can use the function key you want (especially F13 and above) on the Windows system and model you are using.
2. Keys that are assigned to CommandButton controls should not be used for menu items.

Kind:

Do not use the Kind function with this release. It is intended to support merging of menus but there are limitations that prevent it being useful.

Visible:

Specifies whether the menu item is visible.

When an item is not visible other menu items are shifted so that there is no gap in the menu. Any sub-menus of the menu item are also not visible or accessible.

Enabled:

Specifies whether to enable the menu items. The menu items are displayed in gray when they are disabled.

Check Mark:

Specifies whether to display a check mark on the left of the menu item. You cannot specify check marks for top-level menu bar items.

Separator:

Specifies whether this menu item is a separator item. If checked, all other properties are ignored.

If the items following the separator are not visible, so that the separator would become the last item in the menu, the separator is also made not visible.

You cannot specify this property for top-level menu bar items.

Menu Break:

Specifies whether or not the menu item starts a new line or column.

Not MenuBreak - menu item follows the previous menu item.

MenuBreak - menu item starts a new line or column.

MenuBarBreak - for vertical menus specifies that the menu item starts a new column with a bar separating the columns. For items on the menu bar this is the same as MenuBreak.

Editing Menu Items

You can perform the following operations on the selected menu item:

- **Cut:**
Copies the menu item to the clipboard and deletes it from the menu.
- **Copy:**
Copies the menu item to the clipboard.
- **Paste:**
Inserts the menu item copied to the clip board immediately before the selected item.
If the whole menu is selected, the clip board menu item is inserted at the end of the menu.

- **Delete:**
Deletes the selected menu item.
- **Popup Menu:**
Specifies whether the menu being edited is a popup menu.
This option is only enabled when the whole menu is selected - which you can do by clicking in the white space of the Menu Editor window.
If it is checked, the menu is a pop up menu.
If it is not checked, the menu is a menu bar.

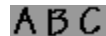
Creating Toolbars

Toolbars consist of mini icons that may be clicked like command buttons (push buttons) to create events.

The images for the mini icons are created as bitmaps (.BMP files) which are typically rectangles of 16 x 15 pixels, but can be any size you choose. For example, here are three sample 16 x 15 pixel bitmaps:



When you want to put these bitmaps into a PowerCOBOL toolbar you combine the individual bitmaps into a single bitmap, referred to as an "image list". The image list is simply a bitmap file with multiple mini icons arranged in sequence from left to right. The image list bitmap looks like this:



It is a simple .BMP file that is 16 pixels in height and 48 pixels in width.

You can create the bitmap files using the Paint utility provided with Windows, or just about any graphics program such as Adobe PhotoShop or Corel PhotoPaint. Remember that they are not really special icons, but rather 16 x 15 pixel pictures (.BMP files).


Once you have all of your individual mini icons (which should be uniform in size), you assemble them into a single .BMP file, the image list, by copying and pasting them. Having prepared your image list you are ready to create your toolbar in PowerCOBOL.

The following section takes you through the steps required to create and use a toolbar in PowerCOBOL.

Creating a Sample Toolbar Application

The following steps describe how to create a simple toolbar application like the one provided with the PowerCOBOL product in the C:\Program Files\FujitsuCOBOL\COBOL\samples\PowerCOBOL\Toolbar folder. You can use the abc.bmp file from that folder.

1. Start PowerCOBOL and create a new Standard Form project by selecting New from the File menu. Select Standard Form in the New Project dialog, and click OK.
2. Right click over the project name ("Untitled") in the left window pane of the Project Manager to display the pop-up menu, and select "Expand All". PowerCOBOL expands the project tree to show all the project components.

3. Include the image list in the project by right clicking on the module name "Main" and selecting "Insert File" from the pop-up menu.
Power COBOL displays the Insert File dialog box.
4. Click on the "Files of Type:" drop down and select "Image List (*.BMP)".
All files with a .BMP extension are displayed.
5. Navigate to the folder containing the file abc.bmp, select it, and click Open.
PowerCOBOL adds an imagelist item under "Main [Module]" with the name "ImageList1" already selected, ready for you to overtype it with your name for the image list.
6. Type "ABC" followed by Enter, to give the name ABC to the image list.
You are now ready to create the toolbar control on the form.
Note that the ImageWidth property, displayed in the right-hand pane, defaults to 16 pixels. As this is the width of the bitmaps that make up the image list we do not have to change it. If you were using images of a different size you would need to edit this figure.
7. Right click on "MainForm [Form]", and select Open to bring up the Form Editor.
8. Insert a toolbar by clicking on the Toolbar control  in the tool palette, moving the mouse over the form, and clicking on the form.
A handle appears at the top of the form.

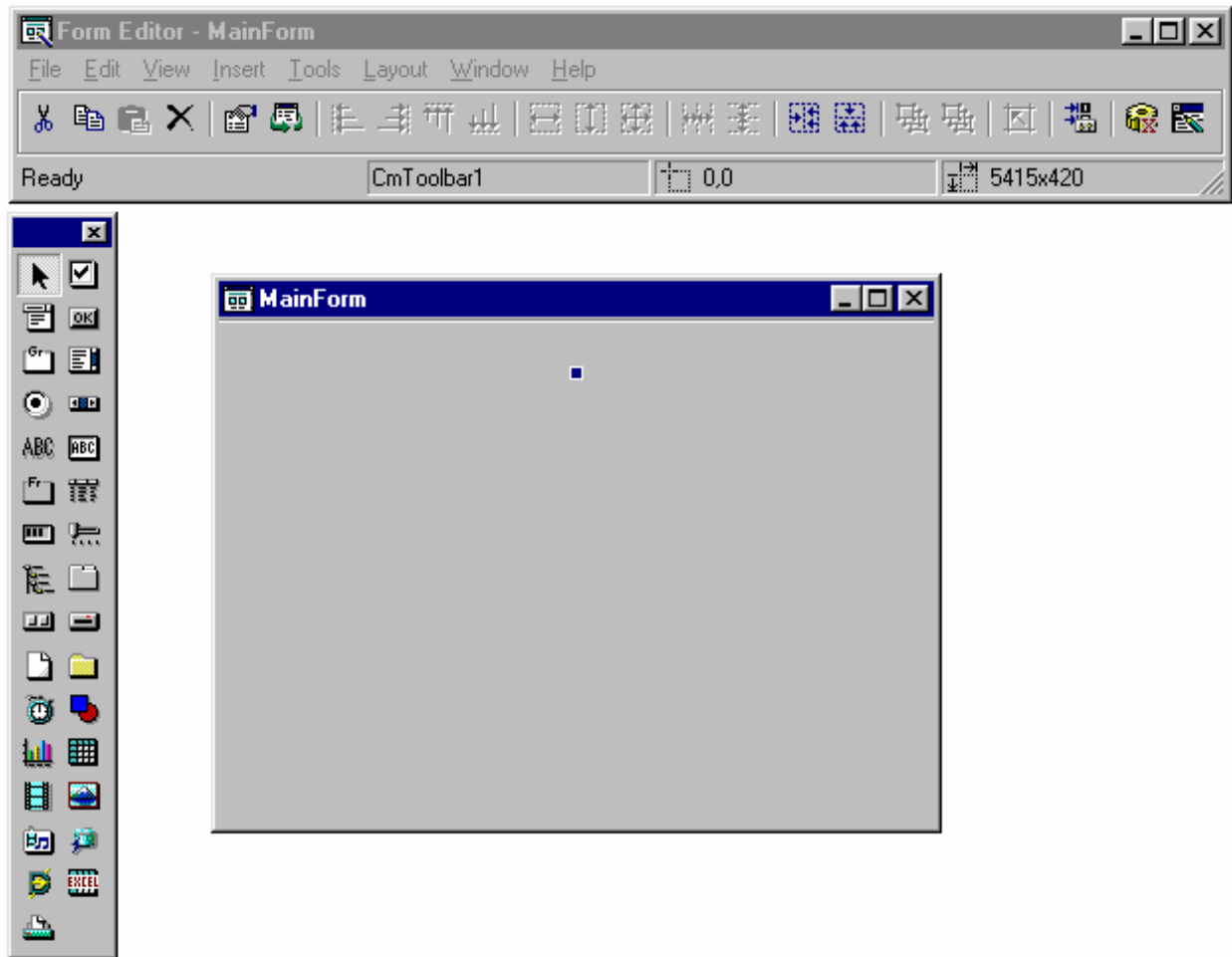


Figure 4.18. A toolbar control on a form.

9. Right click on the (blank) toolbar and select Properties from the pop-up menu. PowerCOBOL displays the toolbar Properties dialog box.
10. In the "Image List" field enter "ABC". This tells PowerCOBOL which image list (ABC.BMP) to use for this toolbar.

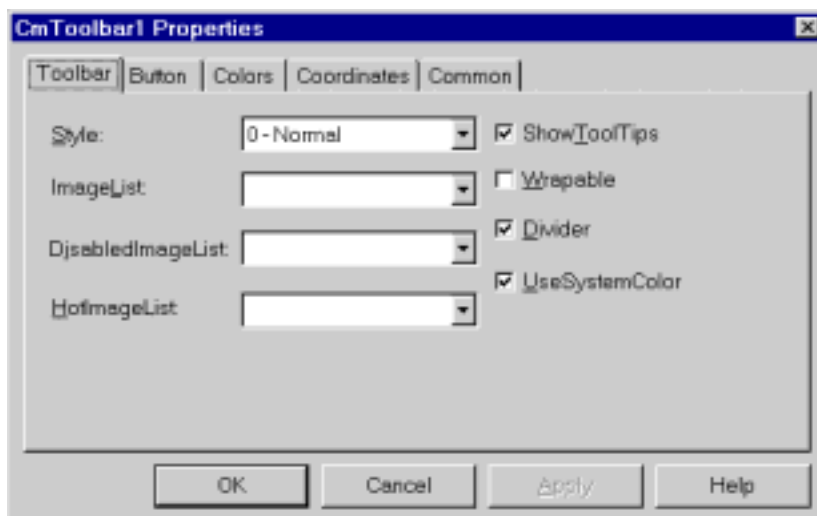


Figure 4.19. Toolbar properties.

11. Now click on the Button tab. The Button tab is where you define the details of each element in the toolbar.
12. Click on the Add command button to define the details for the first toolbar button.
The Index property, which contains the index number of the button, is set to 1. Each item on the toolbar is referenced using an index number (kind of like a subscript). The index numbers correspond to the physical order of the items on the toolbar from left to right. For example, index 1 is the first item from the left; index 3 is the third button from the left, etc.
13. Leave the Style property as "0-Normal" (we will discuss other settings later).
14. The Caption property allows you to specify a caption that is displayed beneath the button. Leave the Caption field blank.
15. The ToolTipText property is extremely useful. It allows you to enter a description of what the push button does. At runtime, when the user moves the mouse over this button, PowerCOBOL displays a "tooltip" containing the description. Our tooltip will be rather basic: enter "A push button" in the ToolTipText property.
16. The ImageIndex property is an index into the image list. PowerCOBOL uses the ImageWidth property of the image list to determine the width of the images contained in the image list. It then logically breaks the image list into separate images, numbering them from the left. You set the ImageIndex property to the number of the image you wish to display on this button.
For the first button, set ImageIndex to 1.
17. Click on the Apply button and you should see a mini icon appear in the toolbar area of the form. Do not click on the OK button yet.
18. We now add the additional mini icons to the toolbar. Note that for purposes of illustrating features, we will create a small space between this first mini icon and the next one. This space is known as a separator in PowerCOBOL. To create this space, click on the Add button again and click on the Style drop down list.
Notice that there are 4 styles available:

- 0-Normal - This creates a normal push button mini icon as you did in the steps above.
- 1-Separator - This creates a separator (a small space between the previous button and the next button).
- 2-CheckBox - This creates a push button that will remain depressed if the user clicks on it. For example, if you have a button that forces a **Bold** text style when typing that you wanted to be able to toggle on and off, this is the setting you would use. When a user clicks on the push button, it will remain depressed to show that whatever it represents is still in effect. When the user clicks on it a second time, it will go back to normal.
- 3-CheckBoxGroup - This allows you to group CheckBoxes into a group so that only one button in the group can be depressed at a time. Depressing one button causes a previously depressed button in the group to be raised. This is useful for situations where you have two or more mutually exclusive conditions, for example, if you had three CheckBoxes that changed the background color of a form to red, green, or blue, respectively.

19. Now back to building the toolbar.

In the Style dropdown, select 1-Separator to create a space between the previous icon and the next icon. Click on the Apply button.

Notice that you did not enter a value in the Image Index property because a separator doesn't require an image. Consequently the button Index and the ImageIndex for the following buttons will not be the same.

20. Click on the Add button again to add the next button.

Leave the Style property as 0-Normal.

Set the ToolTipText property to "B Push Button".

Set the ImageIndex to 2.

21. Click on the Apply button to see your button added to the toolbar. (You don't have to click on the Apply button after every change but it is often helpful to see the effect of your changes.)

22. Add the final button without a separator by clicking Add and setting up:

Style as 0-Normal.

ToolTipText as "C Push Button".

ImageIndex as 3.

23. Click OK.

Your form should now look like this:

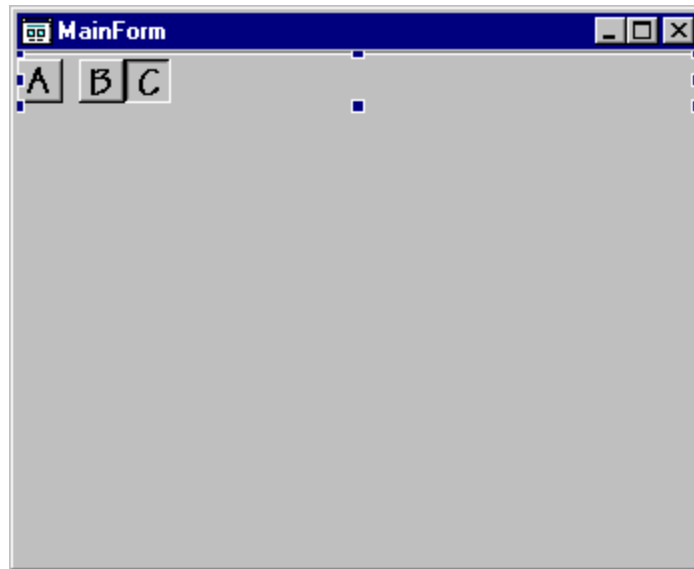



Figure 4.20. A form with a toolbar.

We will now add a static text control to display some text at runtime.

24. In the toolbox palette select the StaticText control .
25. Move the pointer over the form and place the static text control to the left of the form near the toolbar.
26. Use the handles of the static text control to stretch it to about twice its original width.
27. Double click on the static text control to display the Properties dialog.
28. In the Properties dialog box, blank out the Caption property, so that no default text is displayed in the static text control.
29. Now resize the form to match the width and depth of the static text control:

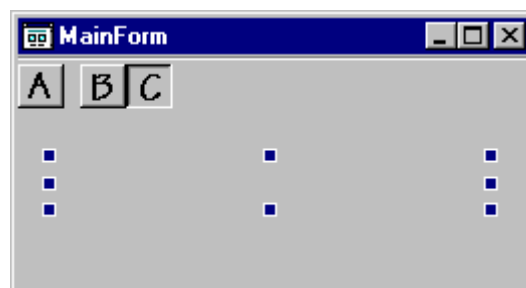


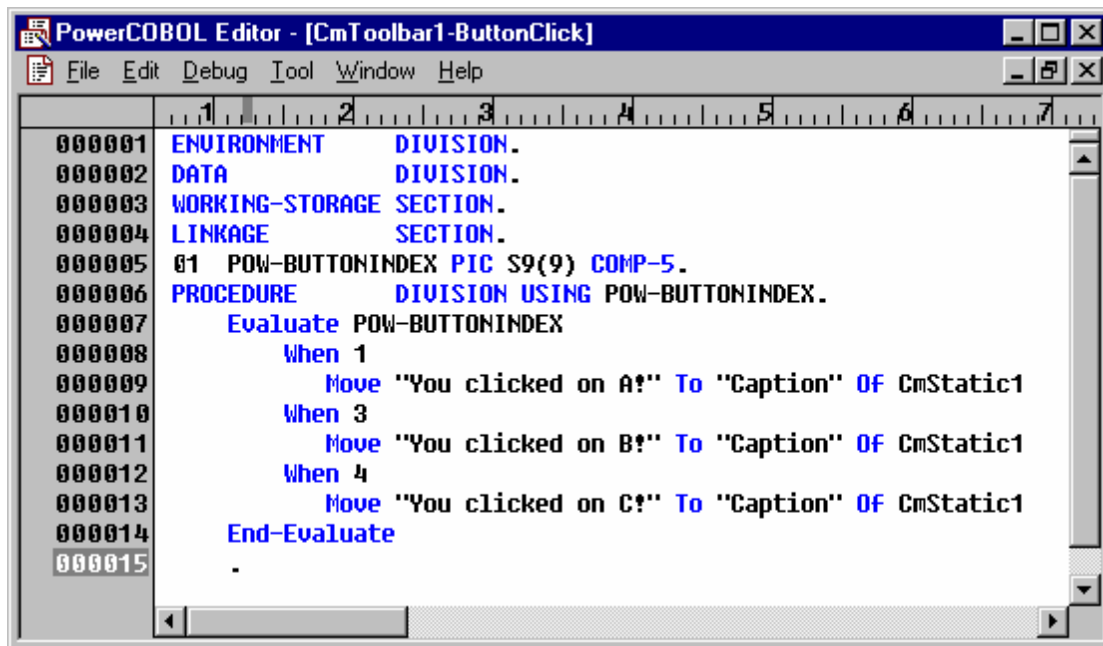
Figure 4.21. The form with a static text box control.

Now it is time to write the event procedure for the toolbar. The behavior we will implement is that whenever a user clicks on one of the buttons in the toolbar, we will display text in the static text control telling which button was clicked.

When a toolbar button is clicked a *toolbar* event, *ButtonClick*, is generated (NOT an event unique to the toolbar button). The same event procedure receives all the button clicks. This procedure is passed the Index number of the button that was clicked and can take the appropriate action.

Also recall that, because we have a separator on the toolbar, the button indexes are 1, 3 and 4 (NOT 1, 2 and 3).

30. Right click on the toolbar and select "Edit the event procedure" from the pop-up menu.
31. Select ButtonClick from the sub-menu.
PowerCOBOL brings up the PowerCOBOL Editor ready to accept code for the ButtonClick event.
Notice that a value named POW-BUTTONINDEX is being passed into this event procedure. This is the index number of the button that was clicked.
32. Enter the COBOL code shown below.



The screenshot shows the PowerCOBOL Editor window titled "PowerCOBOL Editor - [CmToolbar1-ButtonClick]". The window has a menu bar with File, Edit, Debug, Tool, Window, and Help. Below the menu bar is a ruler with markings from 1 to 7. The main text area contains the following COBOL code:

```

000001 ENVIRONMENT    DIVISION.
000002 DATA          DIVISION.
000003 WORKING-STORAGE SECTION.
000004 LINKAGE        SECTION.
000005 01 POW-BUTTONINDEX PIC S9(9) COMP-5.
000006 PROCEDURE      DIVISION USING POW-BUTTONINDEX.
000007     Evaluate POW-BUTTONINDEX
000008         When 1
000009             Move "You clicked on A!" To "Caption" Of CmStatic1
000010         When 3
000011             Move "You clicked on B!" To "Caption" Of CmStatic1
000012         When 4
000013             Move "You clicked on C!" To "Caption" Of CmStatic1
000014     End-Evaluate
000015 .

```

Figure 4.22. Event Procedure code for the Toolbar.

33. Now save the editor session and close it.
34. Return to the Project Manager window.
35. Save the project naming it "Toolbar1".
36. Build the project by right clicking on the project name and selecting the Build All option from the pop-up menu.
If necessary correct any errors and repeat the build.
37. Once you have a clean build, execute the application by right clicking on the Main module and selecting Execute from the pop-up menu.
When the PowerCOBOL Run-time Environment window appears, click on the OK button.
PowerCOBOL displays the toolbar window:

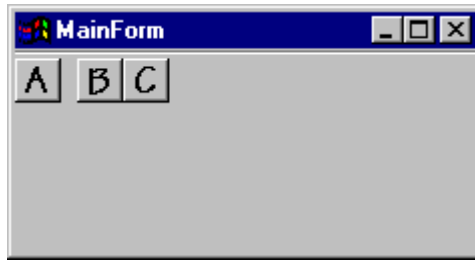


Figure 4.23. The Toolbar application.

38. Click on the toolbar buttons.
You should see a text message appear stating which button was clicked.
39. Hover the mouse over any of the buttons to see the Tooltips.
40. When you are finished testing out the application, close it.

Printing Forms

You can print your form layouts from either the Form Editor or Project Manager:

Printing from the Form Editor Window:

Select Print from the File menu.

Printing from the Project Manager Window:

Right click on the form name in the project tree and select Print from the pop-up menu.

Printing Procedure Code

You can print all your PowerCOBOL event procedures from the Project Manager window or one at a time from the PowerCOBOL Editor window.

Printing from the PowerCOBOL Editor Window:

Select Print from the File menu.

PowerCOBOL prints the current event procedure or external COBOL source.

Printing from the Project Manager Window:

Right click on the COBOL Script group item in the project tree and select "Print procedures" from the pop-up menu.

PowerCOBOL prints all the event procedures.

Chapter 5. Writing Event Procedures

This chapter explains how to write event procedures (COBOL scriptlets) using PowerCOBOL. The following topics are discussed:

- Inserting a control name
- Writing an event procedure
- Inserting properties and methods
- Searching and replacing text strings

Overview

An *event* is an expected action that occurs when a user interacts with available objects in the user interface. For example, the user expects an animation clip to play when the "Start " or "Play" button is pressed and to pause when the "Pause" button is pressed.

Writing a proper Windows GUI application entails creating all of the potential windows (forms) and objects (controls) that the user may interact with, determining all of the possible events that may occur, and creating a link between each potential event and the application code to handle the event. This concept is known as *event-driven programming*.

Writing and editing event procedures, therefore, is the process of determining the possible events for each form and control and then creating the links between the events and the application code handling the events.

Steps in Writing an Event Procedure

Writing an event procedure involves the following steps:

1. Right-click the mouse on the control you want to create an event procedure for and select Edit Event Procedure from the pop-up menu.
2. Choose the action (event) you want to associate with the control from the secondary pop-up menu that appears. The PowerCOBOL Editor window will appear.
3. In the Editor window, 'drag and drop' related controls and write the COBOL statements to describe the actions associated with the selected control or form. You can also select potential properties and methods to be applied to the control or form. See "Writing an Event Procedure" and "Inserting Properties and Methods" for more details on writing event procedures.
4. Save and close the Editor window to save the event procedure.

NOTE

Repeat the above steps to program additional events for a specific control or form.

The Edit Event Procedure Pop-up menu

In the Edit Event Procedure pop-up menu, you determine the event (or events) you want to associate with the selected control or form. Each control has a set of possible events to choose from. For example, a simple animation control has seven possible events: Click, DblClick (double click), EndAnimation (end), MouseDown (push the mouse button down), MouseMove (move the mouse over the animation), MouseUp (let the mouse button up) and StartAnimation (start).

The figure below illustrates these seven possible events for a simple Animation control:

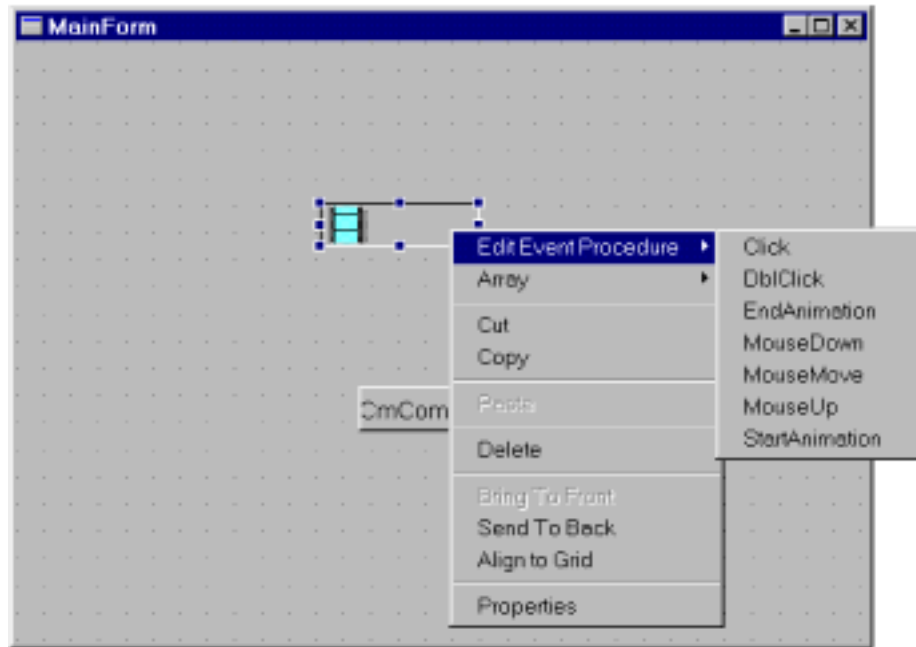


Figure 5.1. Event selection for a simple Animation control

For a description of the possible events for each control and form, refer to the online help system.

After selecting an event, the PowerCOBOL Editor window will appear for this event. You may begin creating COBOL code to handle this event at this point, or modify any existing code.

PowerCOBOL Editor Window

In the PowerCOBOL Editor window, you write COBOL statements and 'drag and drop' related controls and/or forms to create the events associated with the selected control or form. You can also select *properties* and *methods* to be applied to the control or form. A property refers to the modifiable attributes of a control or form (for example, the font size). A method refers to the possible actions that a control or form can perform (for example, moving the control to a new position on the form).

If the event you select has not previously had code written for it, you will be presented with the following editor window:

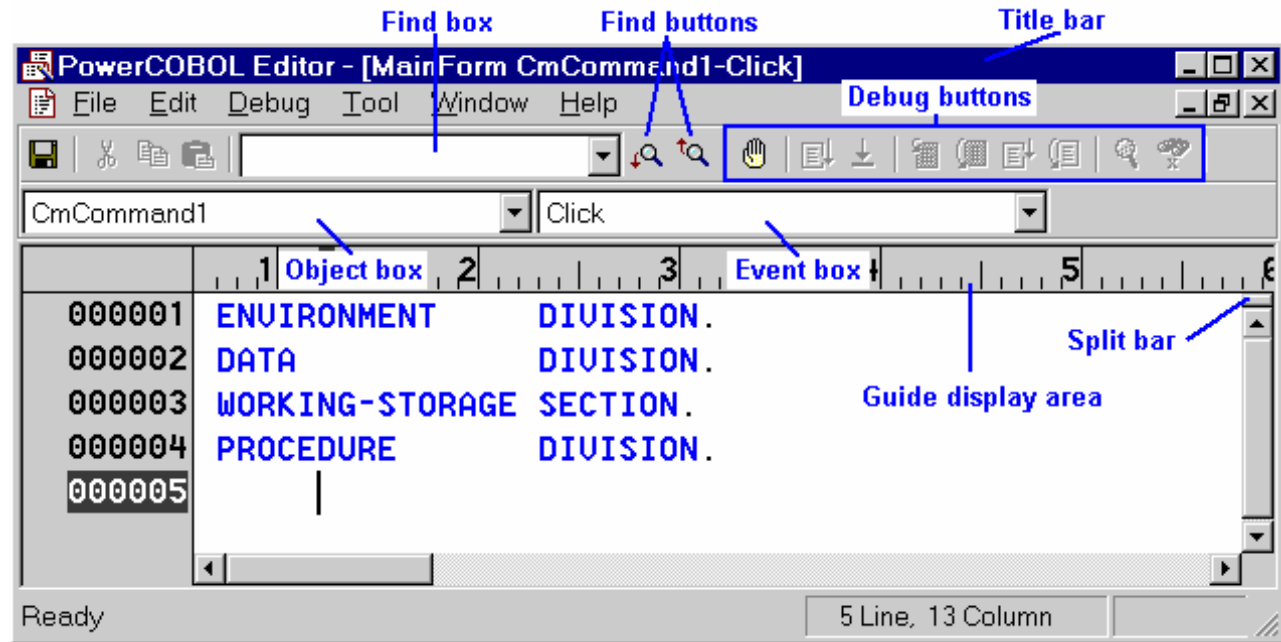


Figure 5.2. The PowerCOBOL Editor window with Debug buttons.

For a complete list of COBOL statements that can be applied to controls and forms as well as a listing of available properties and methods, refer to the "PowerCOBOL Reference".

The PowerCOBOL Editor window contains the following elements:

Title Bar

The title bar displays the name of the control and the name of the event.

Guide Display Area

The guide, displayed across the top of the window below the title bar, acts as a ruler to show the width of columns and the current column position of the cursor. To display or hide the guide, select or de-select Display Column Guide in the Options menu.

Line Number Display Area

Line numbers keep track of how many lines are included in a procedure, and what is on each line. The current line number containing the cursor is also highlighted. Line numbers can also be displayed or hidden by selecting or de-selecting Display Line Number in the Options menu.

NOTE

You can also select whether or not to display the Tab and Carriage Return characters and/or wide blanks, adjust the tab width, indent new lines and be prompted to save the open event procedure in the Options menu.

Find box and Find buttons

Specify a string to find in the Find box, and then click the appropriate Find button to find forward or backward.

NOTE

You cannot specify the "Match whole word only" and "Match case" options in this case. Select the "Find" option in the "Edit" menu if you want to use the options.

Object box and Event box

Select a control or form name in the Object box and an event in the Event box to open and edit the event procedure.

Split bar

You can split the editor window into two panes by using the Split bar as follows:

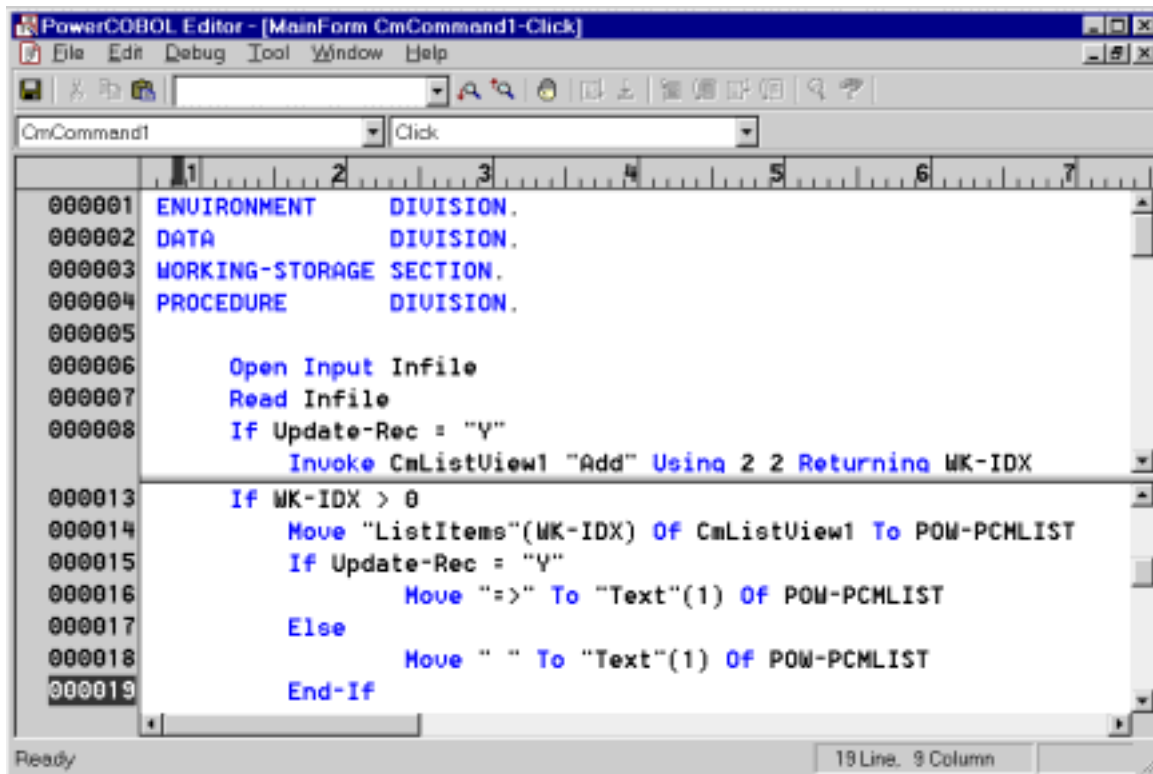


Figure 5.3. The PowerCOBOL Editor in split screen mode.

Move to the top, move to the bottom or double click the Split bar to release the split.

Debug Buttons

When you debug your PowerCOBOL programs, you observe your code using the PowerCOBOL Editor. The Debug buttons are enabled. They cover such functions as stepping into or over the code, executing the code to the next breakpoint, breaking into execution, and watching data items.

Comment Line setting

If you select the "Set Comment Line" option of the "Edit" menu, PowerCOBOL inputs an asterisk (*) into the indicator area (column 7) of the selected lines. If you didn't select any lines, PowerCOBOL inputs an asterisk indicator area into the line that has a caret (the line the cursor is positioned on).

If you select the "Release Comment Line" option, PowerCOBOL inputs a space into the indicator area.

Go to a Line

You can go to a specified line by using the "Go To" dialog box as follows:



Figure 5.4. The "Go To" dialog box.

The PowerCOBOL Editor provides these options and is customizable as well.

NOTE

Most of the editor screen shots in this manual have the editor toolbar turned off, so your editor may appear differently.

Creating and Accessing Event Procedure Code from the Project Manager

You may create new code and access existing event procedure code from the PowerCOBOL Project Manager window, without opening a form for editing.

You can do this in the left windowpane by simply right clicking the mouse on the control whose event procedure code you wish to create or modify.

This brings up a pop-up menu from which you can select Edit Event Procedure. You are then presented with a secondary pop-up menu listing the available events as shown in the following figure:

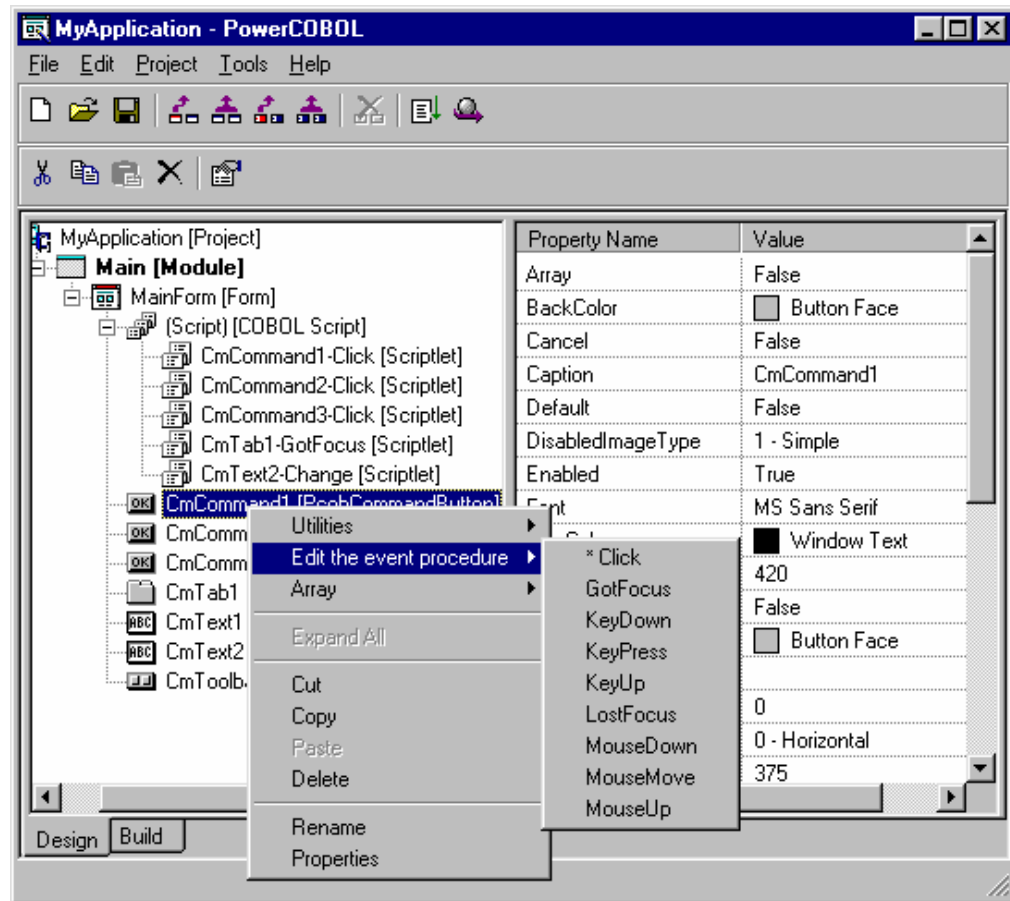


Figure 5.5. Selecting an event procedure from the Project Manager window

NOTE: An asterisk (*) beside an event means that there is already a custom event procedure associated with it.

When you select one of these events, the PowerCOBOL Editor window will appear containing the code for that particular event.

If you examine the left windowpane of an application that already contains some custom event procedure code for one or more controls, you will note a node in the hierarchical tree diagram named [COBOL Script] (see Figure 5.5).

This node will list under it all existing custom event procedures [scriptlets]. You may right-click the mouse on any one of these scriptlets and select Open from the pop-up menu to open it in the Editor for modification.

Navigating within the PowerCOBOL Editor Window

There are two ways to navigate within the Editor window. You can use the mouse or use a combination of keys on the keyboard.

Moving the Cursor Using the Mouse

Moving the mouse directs the pointer on the screen to the position you want.

NOTE: Once the cursor is in position, clicking on either of the mouse buttons trigger actions.

Moving the Cursor Using the Keyboard

To move the cursor using the keyboard, press the following keys or key combinations in order to navigate within the PowerCOBOL Editor window:

HOME	Move to the beginning of a line
CTRL + HOME	Move to the beginning of a procedure
END	Move to the end of a line
CTRL + END	Move to the end of a procedure
PgUp	Move to the previous screen
PgDn	Move to the next screen
Up Arrow	Move up to the previous line
Down Arrow	Move down to the next line
Left Arrow	Move left to the previous character
Right Arrow	Move right to the next character

Inserting/Overwriting Characters

Text can be entered in one of two modes: *insert* mode or *overwrite* mode. In insert mode, the cursor is a blinking vertical line, and as you type, new text pushes existing text forward. In overwrite mode, the cursor is a block cursor, and as you type, new text replaces existing text.

NOTE

When you are in overwrite mode, and press the ENTER key at the end of a line, on the next line, text entry returns to insert mode. Also, when you are in overwrite mode, if the cursor comes to a TAB character, what you type is inserted in front of the TAB character rather than overwriting it. However, if there is no room left to insert characters, the TAB is overwritten. (Try it. You'll like it.)

To switch between the two modes, press the INSERT key. The default is insert mode.

Deleting Characters

Delete characters by pressing the DELETE or BACKSPACE key. The DELETE key erases characters ahead of the cursor. The BACKSPACE key erases characters behind the cursor.

You may additionally use the mouse to select a block to delete. You do this by moving the mouse to the start of the text, holding the left mouse button down and dragging the mouse to the end of the text. Release the mouse button to highlight the block of text. You can then hit the DELETE key to delete the selected block of text.

Indenting Lines Automatically

To line up statements in a procedure, use the Indent feature to place spaces or tabs at the front of a line, by selecting Indent from the Options menu.

When you press the ENTER key at the end of a line and the indent function is active, the cursor is placed on a new line below and is automatically aligned (indented) with the text immediately above.

Inserting a Control Name

In the PowerCOBOL Editor window, controls and forms are referenced using their *object name*. For example, if you want the “Play” button to begin playing a simple animation, the event procedure of the “Play” button must refer to the animation control’s control name (for example, CmAnimation1) in the INVOKE statement. There are three ways to insert a control name: use the ‘drag and drop’ technique, use the menu commands or use keyboard commands.

Using the ‘Drag and Drop’ Technique

In the PowerCOBOL Editor window, move the cursor to the position where the control name is to be inserted. Then move the mouse over the control whose name you wish to insert. Press and hold the left mouse button over the control on the form to select it.

NOTE: For best results, access to the PowerCOBOL Editor window and the open form should be unobstructed so that neither window obscures the other when selected.

Drag the mouse as if you are trying to move the control from the form and drop it onto the Editor window. Release the left mouse button to drop the control in the Editor window.

The name of the control should now appear in the Editor, having been inserted at the point the cursor was positioned.

Using Menu Commands

Click on a control in the open form to select it and then select Copy in the Edit menu to move a copy of the control to the clipboard. Alternatively, you may simply right-click the mouse on the control and select the same Copy function from the pop-up menu.

Now move back to the Editor window. Move the cursor to the position where the control name is to be inserted. Select Paste in the Edit menu to place a copy of the control name at the cursor position. Alternatively, you may right-click the mouse anywhere in the Editor window and select the same Paste function from the pop-up menu.

Using Keyboard Commands

Click on a control in the open form to select it. Select the Copy (CTRL + C) keyboard command.

Move back to the Editor window and move the cursor to the position where the control name is to be inserted. Select the Paste (CTRL + V) keyboard command to place a copy of the control name at the cursor position.

Refer to “Editing a Control” in Chapter 4 for a complete list of key combinations.

Writing an Event Procedure

This section explains how to write an event procedure. It covers the following topics:

- Declaration statements and common procedures
- Form procedures
- Control procedures
- Procedures for controls in arrays
- The #INCLUDE statement
- Notes on programming

Declaration Statements and Common Procedures

In standard COBOL, which is based on English and designed for business applications, programs are structured into four sections referred to as *divisions*. They are:

- IDENTIFICATION DIVISION: names the program and contains optional information to identify the author, when it was written, and its function.
- ENVIRONMENT DIVISION: identifies and describes the files used by the program.
- DATA DIVISION: explicitly describes each data control.
- PROCEDURE DIVISION: contains the executable program statements.

Form Divisions, Sections and Event Procedures

Unlike individual control procedures (scriptlets), which typically contain only local data and execution statements, form procedures also include declaration statements for the entire form and for any common procedures within the project. The kinds of statements that can be included are:

- SPECIAL-NAMES
- REPOSITORY
- FILE-CONTROL
- BASED-STORAGE
- FILE
- WORKING-STORAGE
- CONSTANT
- PROCEDURE

To access form procedures, right-click the mouse on any blank area of a form. The following pop-up menu will appear:



Figure 5.6. Form Procedures Pop-up menu.

From this pop-up menu, you can display individual form divisions, sections, and event procedures by selecting the appropriate option.

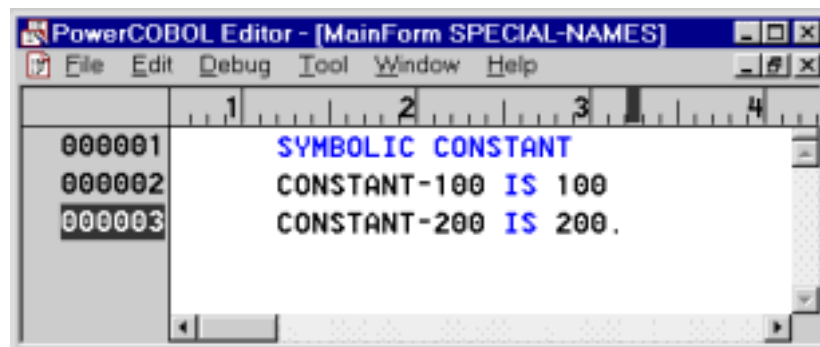
ENVIRONMENT DIVISION

If you move the mouse over Edit ENVIRONMENT DIVISION, a secondary pop-up menu will appear from which you can select SPECIAL-NAMES, REPOSITORY or FILE-CONTROL.

SPECIAL-NAMES

Describes the SPECIAL-NAMES statement in the CONFIGURATION section of the ENVIRONMENT DIVISION.

Declare the name and value of SPECIAL-NAMES such as symbolic constants in the SPECIAL-NAMES event procedure.



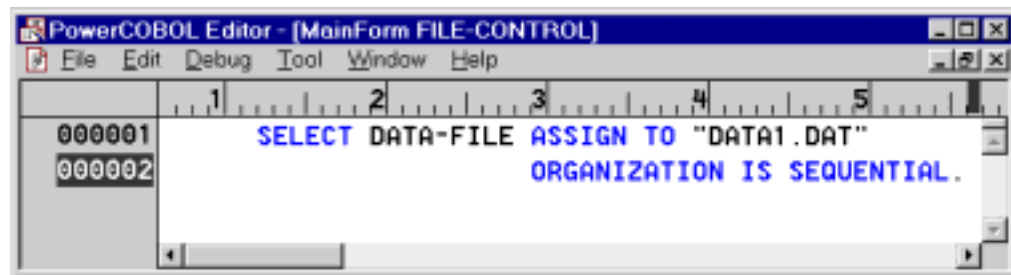
*Figure 5.7. SPECIAL-NAMES declaration***REPOSITORY**

Describes optional Class definitions such as the special *OLE class.

FILE-CONTROL

Describes the FILE-CONTROL statement of the INPUT-OUTPUT section in the ENVIRONMENT DIVISION of a COBOL program.

Use the SELECT statement in the FILE-CONTROL event procedure to declare which files the form will use. This function is required only if a file is called from within a form. The following figure shows how the FILE-CONTROL is declared.

*Figure 5.8. FILE-CONTROL declaration***DATA DIVISION**

If you move the mouse over Edit DATA DIVISION, you can select the BASED-STORAGE, FILE, WORKING-STORAGE, or CONSTANT sections.

BASED-STORAGE

Describes the BASED-STORAGE section in the DATA DIVISION of a COBOL program. BASED-STORAGE is a Fujitsu extension to COBOL85. Refer to the "NetCOBOL Language Reference" for more information.

FILE

Describes the FILE section in the DATA DIVISION of a COBOL program.

Specify the File Descriptor of data files (FD's) and their associated record definitions, in the FILE section. The following figure shows how the FILE section is declared.

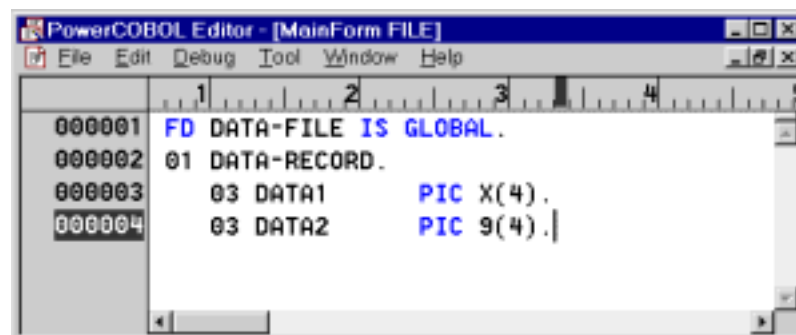


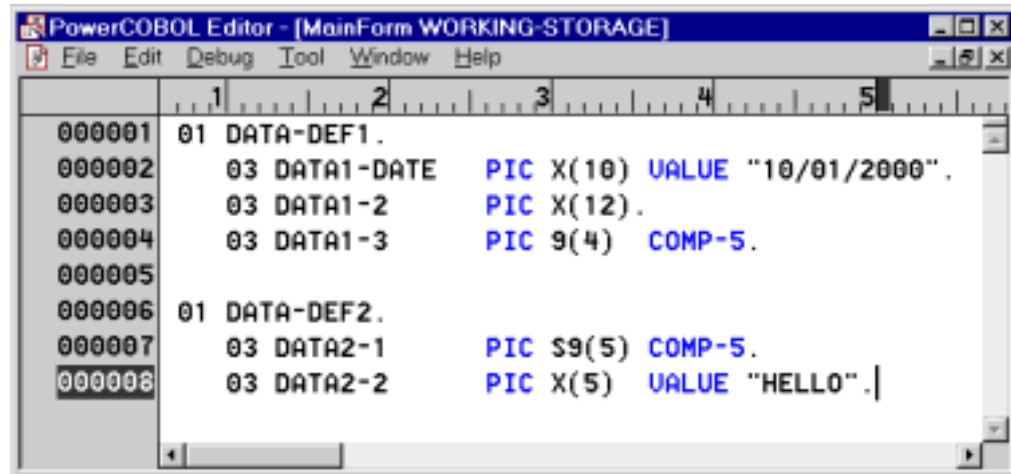
Figure 5.9. FILE declaration

The IS GLOBAL clause in the FD statement allows the file to be accessed from any event procedure (scriptlet) within the form.

WORKING-STORAGE

Describes the WORKING-STORAGE section in the DATA DIVISION of a COBOL program. This is the area where data items are defined.

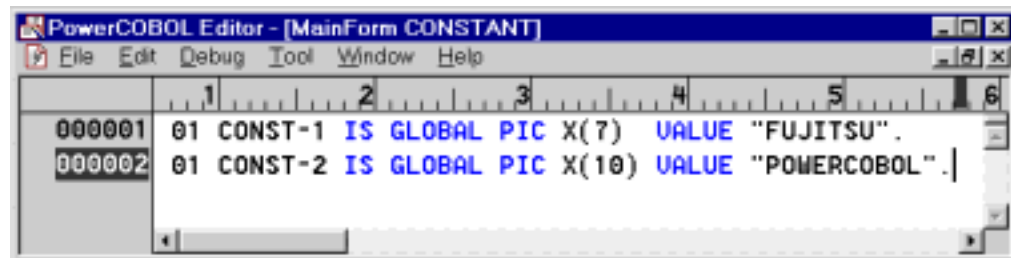
The following figure illustrates how to declare data items in WORKING-STORAGE.

*Figure 5.10. WORKING-STORAGE declaration*

CONSTANT

Describes the CONSTANT section in the DATA DIVISION of a COBOL program.

The following figure illustrates using the CONSTANT statement to declare the names and values that do not change throughout the execution of a program.

*Figure 5.11. CONSTANT declaration*

The use of the IS GLOBAL clause allows the constants to be referenced within any of the form's event procedures.

PROCEDURE DIVISION

If you move the mouse over Edit PROCEDURE DIVISION, you may select the New or PROCEDURE options.

NEW (PROCEDURE)

Allows you to create common internal procedures within a form. These are procedures that are not tied to any specific event, but may be called from other procedures in the application. For example, if you needed to write a procedure to read in some records from a data file and populate a control on the form, you might want to place this code in a separate callable procedure.

Any procedures that are called within a program, or called from within other procedures, can be described in the PROCEDURE window.

PowerCOBOL will assign a default name to each procedure you create here. You may change the name in the left windowpane of the Project Manager window by selecting it and typing in a new name.

Event Procedures Associated with Forms

There are 17 event procedures that are associated with forms:

- Click
- CloseChild
- Closed
- DbClick
- KeyDown
- KeyPress
- KeyUp
- MouseDown
- MouseMove
- MouseUp
- Opened
- PowerBroadcast
- PreKeyDown
- PreKeyPress
- PreKeyUp
- QueryClose
- Resized

See the online help system for details of the above events.

Event Procedures Associated with Controls

Event procedures associated with controls are treated as internal programs. Consequently, the data controls that are defined in the DATA DIVISION of a control's event procedure are applicable only within that procedure.

Internal event data cannot be referenced by other event procedures. If you need to share data or retain data states between individual event procedures, the data items

should be defined as GLOBAL and placed in the form's WORKING-STORAGE SECTION.

If you wish to share data across forms and/or modules, the data items must be defined identically in each form or module's WORKING STORAGE SECTION with the IS EXTERNAL clause added.

You may use the IS GLOBAL or IS EXTERNAL clause with FD's in a Form's DATA DIVISION FILE section to allow data files to be referenced globally or externally as well.

Event Procedures for Controls in Arrays

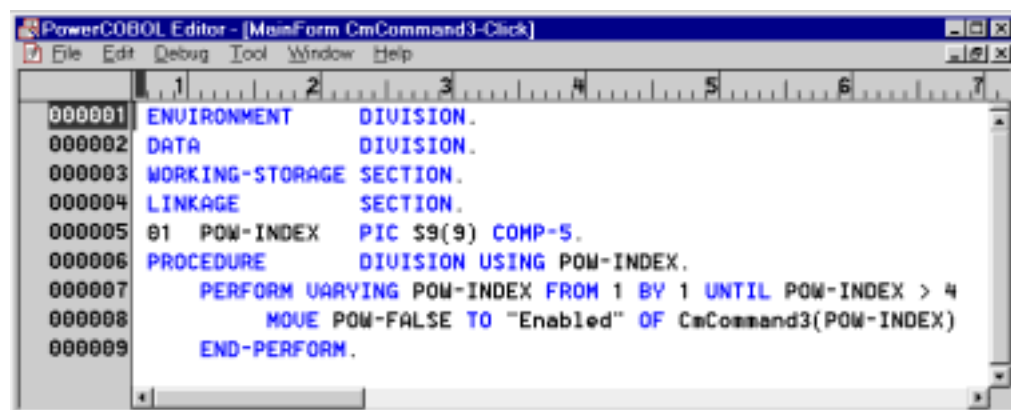
When several controls are placed in an array, one event procedure is created for the array, not one for each control. (Even if an event procedure is meant to be used only on a single control, the event procedure is associated with the entire array.) Event procedures written for controls that are organized into an array can perform operations on all the controls in the array.

Unlike event procedures that are used for single controls, event procedures associated with arrays must include the USING clause in the LINKAGE section and the PROCEDURE DIVISION. The index for the control of the related event is described in the data control in the USING clause.

The control can be located within the array by referencing the data control with an index.

In the following example, four push button controls have been grouped together into an array named CmCommand2.

When any of the four separate buttons is selected by a user, a click event is generated for the entire array. The event procedure code in the example below will actually disable all four push buttons when any one of them is selected (clicked on).



```

PowerCOBOL Editor - [MainForm CmCommand3-Click]
File Edit Debug Tool Window Help
1 2 3 4 5 6 7
000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 LINKAGE SECTION.
000005 01 POW-INDEX PIC $9(9) COMP-5.
000006 PROCEDURE DIVISION USING POW-INDEX.
000007     PERFORM VARYING POW-INDEX FROM 1 BY 1 UNTIL POW-INDEX > 4
000008         MOVE POW-FALSE TO "Enabled" OF CmCommand3(POW-INDEX)
000009     END-PERFORM.
  
```

Figure 5.12. A procedure processing array events

The #INCLUDE Statement

Use the #INCLUDE statement to merge source lines from another file into the source code you are working on at the position of the #INCLUDE statement. The line number of the original statement is replaced with the line number of the statement that results from an operation, such as MOVE.

The #INCLUDE statement is analogous to the COPY statement. The compiler copies the contents of the 'included file-name' into the program at the point where the statement occurs. The contents of the file are treated as additional program source.

The difference between the #INCLUDE and COPY statements relate to the fact that PowerCOBOL uses a preprocessor to turn your program code into an actual COBOL program. #INCLUDE statements are processed by the preprocessor, immediately (during preprocess time), while COPY statements are ignored by the preprocessor and passed onto the COBOL compiler to deal with.

Syntax

```
#INCLUDE "INCLUDE FILE-NAME"
```

Syntax and Usage Rules

- The #INCLUDE statement can be described in the A or B area of the DATA DIVISION, ENVIRONMENT DIVISION and the PROCEDURE DIVISION.
- You cannot use the #INCLUDE statement if the same file name is referenced in an INCLUDE statement in the target file.
- The 'Include file-name' must be a file name in the #INCLUDE library.

General Rules

- Use of the #INCLUDE statement must follow the COBOL language specification.
- The 'included' file can contain #INCLUDE statements.
- #INCLUDE statements can be nested up to 16 levels.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
* Get the #INCLUDE file to the data control
#include "NUMDATA.COB".
PROCEDURE DIVISION.
    MOVE WK-1 TO POW-FONTSIZE OF LABEL1.
    :

+--- The contents of NUMDATA.COB file -----+
|
|000100 01 WK-G.
|000200    02 WK-D1 PICTURE IS S9(4) USAGE IS DISPLAY.
|000300    02 WK-D2 PICTURE IS S9(4) USAGE IS DISPLAY.
|000400 02 WK-1 PICTURE IS S9(9) USAGE IS BINARY.
|000500 02 WK-2 PICTURE IS S9(9) USAGE IS PACKED-DECIMAL.
|
+-----+
```

Notes on Writing Event Procedures

1. **PROGRAM-STATUS**
Use the special register PROGRAM-STATUS with caution. The value of PROGRAM-STATUS is changed whenever a control property is set or whenever a *method* is used.
2. **Data definition in the WORKING-STORAGE section**
To avoid a "USER WORD 'data name' UNDEFINED" error at compile time, the GLOBAL statement must be used to define data in the WORKING-STORAGE, CONSTANT and FILE sections of the form. This is because data defined in these sections are used throughout the program, not just in specific procedures.
3. **The COMMON attribute**
Common procedures are treated like internal programs. For this reason, you must use the COMMON attribute with the PROGRAM clause to describe common procedures within a form.
4. **User-defined words**
No user-defined word can begin with the letters 'POW-' or 'POWER-' when using PowerCOBOL.
5. **The CHANGE event of an Edit control**
The CHANGE event of the Edit control is invoked for each character input to the control, not for the entire string. For instance, if the input character contains four characters, the change event procedure will be invoked four times, once for each character.
6. **The COBOL85 presentation file**
Make the procedure that handles the presentation file a common procedure and make sure that the focus on the presentation file is not interrupted once the OPEN statement is issued and before the END statement is issued.
7. **Ending a procedure**
Never use the STOP RUN statement to close a PowerCOBOL application. It will end the program without removing the open resources from memory.
8. **An infinite loop in a procedure**
An infinite loop is a set of commands that repeat indefinitely. For example, if a

string is set to have a POW-TEXT property within a CHANGE event, the CHANGE event will repeat again and again. In that case, the application usually exits abnormally because of a 'stack violation' error. Alternatively, you can select the CLOSE command from the COBOL control menu. Restart Windows to free up the resources left in memory.

9. INCLUDE files

Use as few INCLUDE files as possible because of the impact they have on debugging and compiling.

10. Using '&' in text or in a title

Because the '&' has a special meaning, (it indicates a shortcut key), it must be 'escaped' to display it in a text or title. Type '&&' to make the '&' symbol display in the following controls; labels, push buttons, radio buttons, check buttons, group boxes, function keys, and bitmap buttons. For example, to display 'NAME & ADDRESS' use:

```
MOVE "NAME && ADDRESS" TO POW-TEXT OF LABEL1.
```

Inserting Properties and Methods

A *property* refers to the attributes of a control or form that can be modified. For example, you can change a text string and the text color when the user clicks on a button.

A *method* refers to the possible actions that a control or form can perform. An animation control, for example, has four possible actions: Move, PauseAnimation, PlayAnimation, and Refresh.

NOTE: Not all controls have methods and a single method can be associated with more than one control.

For a complete listing of the properties and methods available for each control, refer to the "PowerCOBOL Objects Programmer's Guide."

Properties and methods are accessed by highlighting a control in the Editor window and then selecting either Property or Method in the Edit menu.

You may alternatively select an available property or method for a control by highlighting it in the Editor window and right clicking on it with the mouse. This will bring up a pop-up menu from which you can select Properties or Methods.

Inserting a Property

You may insert a property in the Editor by highlighting the name of the control whose property you wish to manipulate.

Then right-click the mouse on the highlighted control name and select Insert Property from the pop-up menu (or select Property from the Edit menu).

A second pop-up menu will appear that lists the available properties for the highlighted control.

Select the name of the property you wish to manipulate. The property name will be placed to the left of the highlighted control name in the proper syntax.

You may then add in a COBOL MOVE statement and include the value you wish to move to the property to set it.

Control and Form Property Syntax Format

The following format is a PowerCOBOL statement used to define the property of a control or a form.

```
[ " | ' ] PROPERTY-NAME [ " | ' ] [ (INDEX-1 ... ) ]
[ OF [ " | ' ] PROPERTY-NAME [ " | ' ] [ (INDEX-1 ... ) ] ]
[ OF CONTROL-NAME [ (INDEX-2) ] ]
OF { CONTROL-NAME [ (INDEX-2) ] | POW-SELF }
```

The components of a property definition statement are:

PROPERTY-NAME

There is a pre-defined list of possible properties for each control in PowerCOBOL. For example, the property name for the date style of the displayed TextBox control (named "CmText1") would be written as follows:

```
"DateStyle" OF "RenderText" OF CmText1
```

or

```
"DATESTYLE" OF "RENDETEXT" OF CMTEXT1
```

You are not required to write it case-sensitive.

NOTE: If the property name is a COBOL reserved word (e.g. Size, Column, etc.), you must write it using quotation marks. For example, the following is incorrect syntax.

```
SIZE OF FONT OF CMTEXT1 => Error
```

For a complete list of properties, refer to the "PowerCOBOL Reference" (online help system).

INDEX-1

When a control has more than one property, each property is identified by an index number or letter combination. The property values can be specified using a literal, for example "TableCells" (1 1) or a variable, for example "TableCells(a b)".

INDEX-2

If controls are organized into an array, a second index number can be used to reference the controls.

CONTROL-NAME | POW-SELF

This is a unique reference to a control or a form. POW-SELF indicates the current form. A few rules must be observed when naming controls and forms to avoid compilation errors:

- Controls must have unique names within a form.
- Forms must have unique names within a module.
- The form name must be unique within an application.
- Names must observe COBOL rules for user-defined names, less than 31 characters long. Refer to the "NetCOBOL Language Reference" for details.
- You cannot select a control or form name that is the same as a form's property or method name.
- You cannot define a data name that is the same as a control or form name.

Examples

Standard: "PROPERTY-NAME" OF CONTROL-NAME

Set the string "sample1" to the caption property of the CommandButton control named "Command1".

```
MOVE "sample1" TO "Caption" OF Command1
```

Control in array: "PROPERTY-NAME" OF CONTROL-NAME(INDEX)

Set the string "sample2" to the caption property of the third OptionButton control named Option2.

```
MOVE "sample2" TO "Caption" OF Option2(3)
```

Control in GroupBox: "PROPERTY-NAME" OF CONTROL-NAME OF GroupBox-control-name

Set the string "sample3" to the caption property of the CheckBox control named Check3 in the GroupBox control named Group1.

```
MOVE "sample2" TO "Caption" OF Check3 OF Group1
```

Setting and Referencing Control and Form Properties

The following COBOL statements can act on the properties that are assigned to controls and forms:

- MOVE
- ADD
- SUBTRACT
- COMPUTE
- IF
- DISPLAY
- EVALUATE

There are some restrictions on the syntax and usage of these statements when they are used to set and refer to properties.

MOVE Statement

The MOVE statement copies the value of a control's property to a data control.

Syntax

```
MOVE CONTROL-PROPERTY TO RECEIVE-CONTROL
```

Syntax and Usage Rules

- The 'receive-control' must be a data control.
- Only one 'receive-control' can be identified per MOVE statement.
- The CORRESPONDING statement cannot be used with a MOVE statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

Move the current value of the horizontal scroll bar control (CmScroll1) to the variable defined in the WORKING-STORAGE section.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 CURR-SCROLL-VALUE PIC S9(9) COMP-5.
  :
PROCEDURE DIVISION.
  :
    MOVE "Value" OF CmScroll1 TO CURR-SCROLL-VALUE.
```

NOTE

Every property has attributes. In the above example, the attribute for Value is S9(9) COMP-5.

ADD Statement

The ADD statement is used to increase the numeric value of one control by the numeric value of another.

Syntax

```
ADD CONTROL-PROPERTY TO {DATA-NAME [ROUNDED]}
    [ON SIZE ERROR UNCONDITIONAL-STATEMENT]
    [NOT ON SIZE ERROR UNCONDITIONAL-STATEMENT]
    [END-ADD]
```

Syntax and Usage Rules

- No more than one control can precede the TO statement.
- The GIVING statement cannot be used with the ADD statement.
- The CORRESPONDING statement cannot be used with the ADD statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

ADD the large step of vertical scroll bar control (CmScroll2) to the CURR-VALUE, which is maintained in the WORKING-STORAGE section.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 CURR-VALUE PIC S9(9) COMP-5 VALUE IS 0.
  :
PROCEDURE DIVISION.
  :
    ADD "LargeStep" OF CmScroll2 TO CURR-VALUE.
  :
```

SUBTRACT Statement

The SUBTRACT statement is used to decrease the numeric value of one control's property by the numeric value of a data control.

Syntax

```
SUBTRACT CONTROL-PROPERTY  
        FROM {CONTROL-NAME [ROUNDED]} ...  
        [ON SIZE ERROR UNCONDITIONAL-STATEMENT ]  
        [NOT ON SIZE ERROR UNCONDITIONAL-STATEMENT ]  
        [END-SUBTRACT]
```

Syntax and Usage Rules

- No more than one control can precede the FROM clause.
- The GIVING statement cannot be used with the SUBTRACT statement.
- The CORRESPONDING statement cannot be used with a SUBTRACT statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

SUBTRACT the large step of the vertical scroll bar control (CmScroll2) from the CURR-VALUE, which is maintained in the WORKING-STORAGE section.

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
    01 CURR-VALUE PIC S9(9) COMP-5 VALUE IS 0.  
:  
PROCEDURE DIVISION.  
:  
    SUBTRACT "LARGESTEP" OF CmScroll2 FROM CURR-VALUE.  
:
```

COMPUTE Statement

The COMPUTE statement is used to calculate the numeric value of a control using the numeric value of another.

Syntax

```
COMPUTE Unique-Data-Name = Arithmetic-formula
```

Syntax and Usage Rules

- A property may be included in an Arithmetic-Formula.
- Only a single data name (i.e. "Unique-Data-Name") may be specified in the receiving field.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

Set the font size of StaticText control (CmStatic1) to be 2 points larger than the font size of CommandButton control (CmCommand1).

```
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
    ...
    COMPUTE "Size" OF "Font" OF CmStatic1 = "Size" OF "Font" OF
CmCommand1 + 2
```

IF Statement

The IF statement is used to evaluate properties and control the execution of the program based on the results.

Syntax

```
IF CONTROL-PROPERTY COMPARISON OPERATOR
    RIGHT-SIDE-OF-CONDITION
THEN {{STATEMENT-1} ... | NEXT SENTENCE }
{
    {ELSE {STATEMENT-2} ... [END-IF]} |
    {ELSE NEXT SENTENCE } |
    {END-IF}}
```

Syntax and Usage Rules

The property of a control can be used as the left side of an IF statement. Refer to the "NetCOBOL Language Reference" for a complete list of the rules for using IF statements.

Example

If the CheckBox control (CmCheck1) is checked then the image called 'IMAGE-BMP' is displayed. If it is not checked then nothing is displayed.

```
IF "Value" OF CmCheck1 = 1
    MOVE "IMAGE-BMP" TO "ImageName" OF CmImage1
END-IF.
```

DISPLAY Statement

The DISPLAY statement is used to display data on the screen of the computer.

Syntax

```
DISPLAY {UNIQUE-NAME | CONSTANT | CONTROL-PROPERTY}
    [UPON CALL-NAME]
```

Syntax and Usage Rules

Refer to the "NetCOBOL Language Reference" for a complete list of rules for using the DISPLAY statement.

Example

Display the image name of image control (CmImage1) on the console window.

```
DISPLAY "Current displayed image is "  
       "ImageName" OF CmImage1.
```

EVALUATE Statement

You can use the properties of a control as selection criteria in the EVALUATE statement.

Syntax

```
EVALUATE  CONTROL-PROPERTY  
WHEN      OBJECT-TO-SELECT  
...  
WHEN      OBJECT-TO-SELECT  
...  
:
```

END-EVALUATE

Syntax and Usage Rules

- You can use the EVALUATE statement to set the properties of a control based on the condition under evaluation.
- The ALSO statement cannot be used with an EVALUATE statement.

Refer to the "NetCOBOL Language Reference" for a complete list of rules for using the EVALUATE statement.

Example

```
EVALUATE  "Caption" OF COLOR-NAME  
WHEN      "BLACK"  
    MOVE  "BLACK "          TO "Caption" OF COLOR-TEXT  
WHEN      "RED"  
    MOVE  "RED "           TO "Caption" OF COLOR-TEXT  
WHEN      "YELLOW"  
    MOVE  "YELLOW"         TO "Caption" OF COLOR-TEXT  
WHEN      OTHER  
    MOVE  "INVALID COLOR" TO "Caption" OF COLOR-NAME  
END-EVALUATE
```


Manipulating Control Properties

The following statements can also be used to change a control's property:

- MOVE
- ADD
- SUBTRACT
- COMPUTE

No other statements can be used. The following sections contain descriptions of these statements and how to use them to change the properties of a control.

MOVE Statement

Use the MOVE statement to change the value of a control's property.

Syntax

MOVE SEND-CONTROL **TO** CONTROL-PROPERTY

Syntax and Usage Rules

- The 'send-control' must be a constant or a data name.
- Only one 'control-property' can be identified per MOVE statement.
- The CORRESPONDING statement cannot be used with a MOVE statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

Change the displayed text of the string display control (TEXT-STRING) into "PowerCOBOL".

```
MOVE    "PowerCOBOL"    TO    "Caption" OF TEXT-STRING.
```

ADD Statement

Use the ADD statement to increase the numeric value of a control's property.

Syntax

ADD {CONSTANT | DATA-NAME} **TO** CONTROL-PROPERTY
[**END-ADD**]

Syntax and Usage Rules

- No more than one control can come before or after the TO statement.
- The ON SIZE ERROR and NOT ON SIZE ERROR clauses cannot be used with the ADD statement.
- The GIVING statement cannot be used with the ADD statement.
- The CORRESPONDING statement cannot be used with the ADD statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

The TIMER event procedure of the timer control.

```
      :  
PROCEDURE DIVISION.  
      :  
      ADD 10 TO "Interval" OF CmTimer1.  
      IF "Interval" OF CmTimer1 > 100  
          Move 10 To "Interval" OF CmTimer1  
      END-IF.
```

SUBTRACT Statement

Use the SUBTRACT statement to decrease the numeric value of a control's property.

Syntax

```
SUBTRACT DATA-NAME FROM CONTROL-PROPERTY  
      [END-SUBTRACT]
```

Syntax and Usage Rules

- No more than one control can come before or after the FROM statement.
- The ON SIZE ERROR and NOT ON SIZE ERROR statements cannot be used with the SUBTRACT statement.
- The GIVING statement cannot be used with the SUBTRACT statement.
- The CORRESPONDING statement cannot be used with the SUBTRACT statement.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

The TIMER event procedure of the timer control.

```
      :  
PROCEDURE DIVISION.  
      :  
      SUBTRACT 10 FROM "Interval" OF CmTimer1.  
      IF "Interval" OF CmTimer1 < 100  
          Move 100 To "Interval" OF CmTimer1  
      END-IF.
```

COMPUTE Statement

The COMPUTE statement is used to calculate the numeric value of a control using the numeric value of another.

Syntax

```
COMPUTE Unique-Data-Name = Arithmetic-formula
```

Syntax and Usage Rules

- A property may be included in an Arithmetic-Formula.
- Only a single data name (i.e. "Unique-Data-Name") may be specified in the receiving field.

Refer to the "NetCOBOL Language Reference" for additional information.

Example

Set the font size of Static text control (CmStatic1) to be 2 points larger than the font size of CommandButton control (CmCommand1).

```
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
    ...
    COMPUTE "Size" OF "Font" OF CmStatic1
        = "Size" OF "Font" OF CmCommand1 + 2
```

Inserting Methods

You may insert a method in the Editor by highlighting the name of the control whose method you wish to manipulate.

Right-click the mouse on the highlighted control name and select Insert Method from the pop-up menu (or select Method from the Edit menu).

A second pop-up menu will appear that lists the available methods for the highlighted control.

Select the name of the method you wish to manipulate. The method name will be placed to the left of the highlighted control name in the proper syntax using a COBOL INVOKE statement (similar to a COBOL CALL statement). Any required parameters will be noted as well for you to fill in.

For example, if you wish to create a Move method for a PowerCOBOL Image Control named CmImage1, highlight CmImage1 in the Editor (or type in the string "CmImage1" if it is not present). You may alternatively drag the control named CmImage1 from the form into the Editor window to have its name inserted at the cursor position.

Once the control name has been highlighted in the Editor, right-click the mouse on it and select Insert Method from the pop-up menu (or select Method from the Edit menu).

Select the method you wish to insert (in this example, the "Move" method). The current line in the Editor will be modified as follows:

```
INVOKE CmImage1 "Move" USING Left Top [Width] [Height]
```

You need to ensure that the parameters for Left and Top contain the left and top screen coordinates where you want the image moved to on the screen. The optional image "Width" and "Height" parameters may be specified as well. The brackets placed around these parameters indicate that they are optional parameters.

Control and Form Method Syntax Format

The following example is a PowerCOBOL statement used to define the method for a control or a form.

```
INVOKE {CONTROL-NAME METHOD-NAME [ (INDEX-1) ] |  
       FORM-NAME }  
[PARAMETER ...]
```

The components of this method definition statement are:

control-name

Identifies the name of the control to be manipulated.

method-name

Identifies the *method* to be used.

parameter ...

Identifies the parameters passed to the *method*.

Return value

The return value is stored in PROGRAM-STATUS.

Index-1

If controls are arrayed, specify an index value to reference each control.

Numbers or variables can be used as the index value.

Searching and Replacing Text Strings

PowerCOBOL provides two separate mechanisms for finding and replacing text in your COBOL event procedures (scriptlets). The first is the find and replace facility in the COBOL editor. This allows you to edit an existing event procedure and do a find and/or replace on any text string in that specific program. It will not however, search past the current program if a string is not found.

The second mechanism is the Project Manager's global find and replace utilities, which will find and/or replace strings in all event procedures in a project for a single search. We will examine both of these facilities briefly here.

Using the PowerCOBOL Editor Find and Replace

You can locate and substitute characters or strings during edit sessions by using the Find and Replace commands. Select Find in the Edit menu to access the Find dialog box. You may optionally type a string in the Editor toolbar's find box and click on the toolbar find forward and backward buttons.

Find Dialog Box

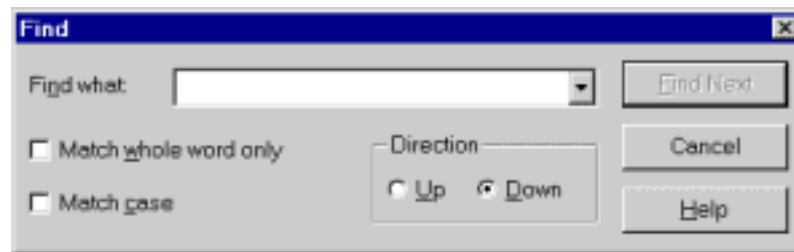


Figure 5.13. The Find dialog box

The Find dialog box contains the following elements:

Find what

Enter the character or string to be searched for in this field.

Direction

Select the direction in which to conduct the search. Select the Down button to search from the starting position to the end of the procedure. Select the Up button to search from the starting position back to the beginning of the procedure.

Match Whole Word Only

Specify that the search is for the strings that match the entire string as a word and not as part of a longer word.

Match Case

Specify that the search is for the strings that match the sequence of upper and lower case letters in the original string.

For example, if you specify this option and are searching for the following string, 'aB', AB would not meet the search criteria.

Click on the Find Next button to initiate the search.

The cursor will move to the location in the current edit session where the string was found, or you will receive in informational dialog box indicating that the string was not found.

Replace Dialog Box

You can find *and* replace characters or strings using the Replace command. Select Replace in the Edit menu to access the Replace dialog box.

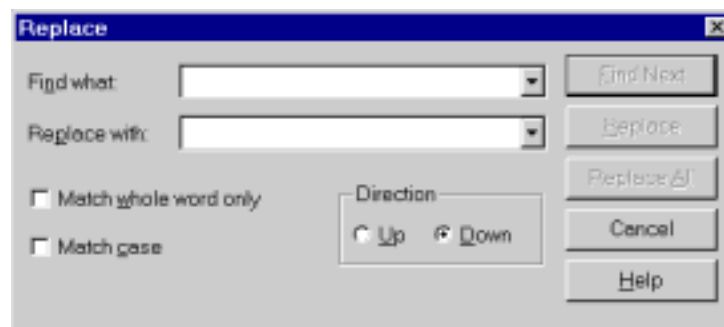


Figure 5.14. The Replace dialog box

The Replace dialog box is similar to the Find dialog box; however, it contains the following additional element:

Replace with

Enter the character or string that is to replace the original character or string. The maximum string length is 32 characters.

Click on the Find Next button to find the next occurrence of the string without replacing it.

Click on the Replace button to find the next occurrence and replace it with the new string.

Click on the Replace All button to find and replace all occurrences of the string.

The system will display a dialog box indicating your results.

Using the Project Manager's Global Find and Replace Utilities

The PowerCOBOL project manager offers functions to find and replace text across all the event procedures and external COBOL procedures in the project or within part of the project.

To find or replace text in part of the project select the node on the project tree that contains the part(s) you want to search. You can then access the Find and Replace functions from:

- The Project menu (Find All/Replace All)
- The Edit menu (Object, Utilities, Find/Replace)
- The pop-up menu (Utilities, Find/Replace)

The sections below explain the global find and replace functions in detail.

To search within a single procedure use the PowerCOBOL Editor Find/Replace functions.

Finding Text

The Project Manager Find (All) function searches for the specified string through event procedures and COBOL procedures belonging to the selected part of the project tree. The Find dialog, in the process of performing a find, is shown below:

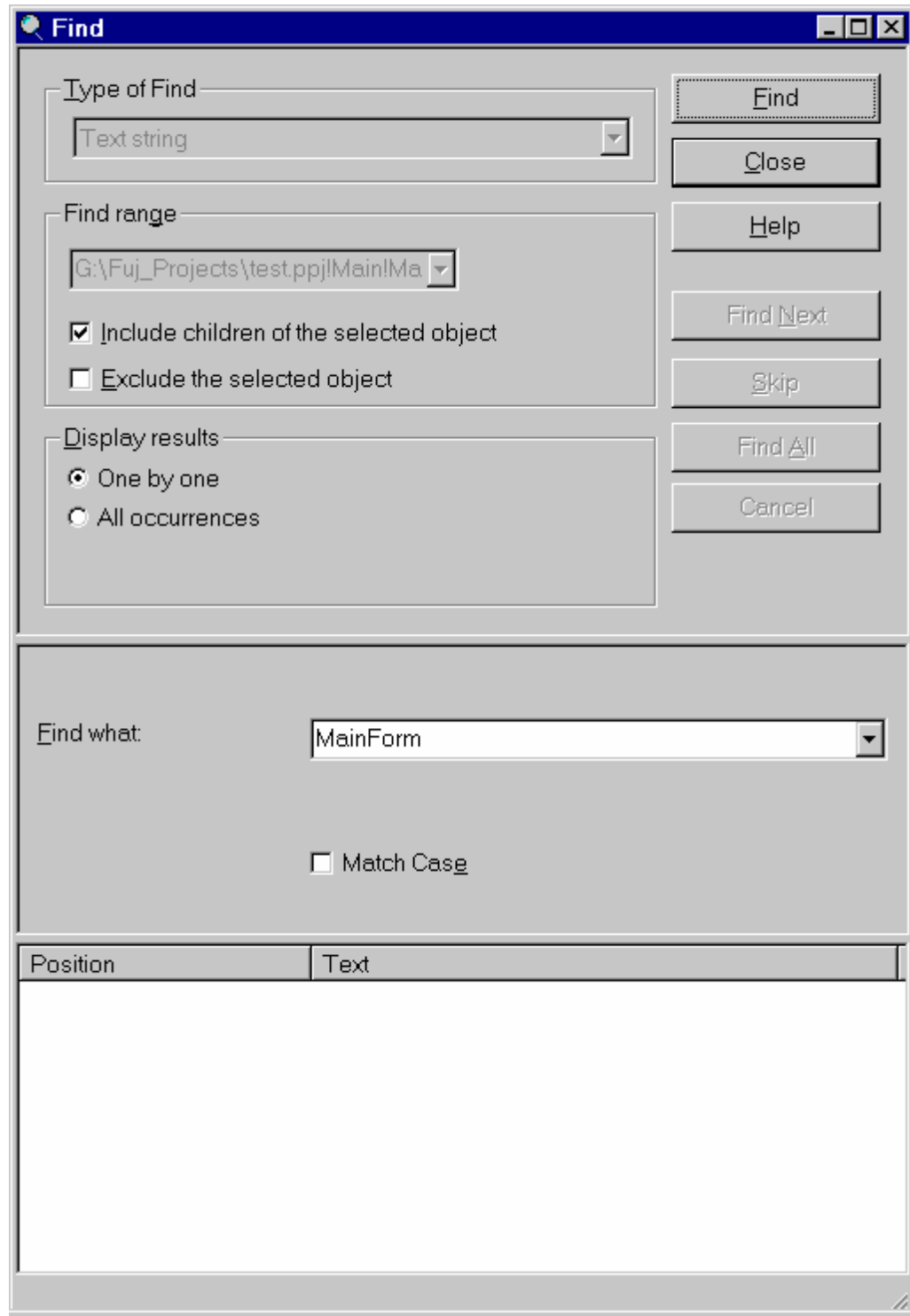


Figure 5.15. The Find dialog window

The Find dialog is made up of three panes: the Control pane, the Target pane, and the Result pane. The contents of these panes are described below.

Control Pane

The control pane provides the functions for specifying the type and range of the Find and for controlling the Find operation. You specify the find range and method of displaying results before you start the Find operation.

Type of Find:

Currently there is only one type of find supported – searching for text – so the selection is disabled. This will be expanded in future releases.

Find Range:

The drop down list shows the name of the project being searched. This field is disabled.

Includes children of the selected object:

Check this box to have Find search objects that are on branches of the selected object in the project tree.

Excludes the selected object:

Select this box if you only want to search objects that are on branches of the selected object and not the selected object. The check box is disabled if “Includes children of the selected object” is not selected.

Display results – One by one:

Select this button if you want Find to stop each time it makes a find.

Display results – All occurrences:

Select this button if you want Find to search through the whole range, displaying results in the Result pane.

Find Button:

The Find button initiates the Find operation, locating the first string that matches the “Find what” string. The Find operation continues until all matching strings have been found or until the Cancel button is pushed.

Close Button:

The Close button closes the Find dialog. It is disabled during a Find operation.

Find Next Button:

The Find Next button is enabled when a Find one-by-one operation starts. It registers the result of the previous Find in the Result pane and searches for the next occurrence of the “Find what” string.

Skip Button:

The Skip button is enabled when a Find one-by-one operation starts. It does NOT register the result of the previous Find in the Result pane and searches for the next occurrence of the “Find what” string.

Find All Button:

The Find All button is enabled when a Find one-by-one operation starts. It causes the Find operation to search for all remaining occurrences of the “Find what” string and display the results in the Result pane.

Cancel Button:

The Cancel button is enabled when a Find one-by-one operation starts. It terminates the Find operation so that you can start a new Find operation or close the Find dialog.

Target Pane**Find what:**

Specify the string for which Find should search in the "Find what" combo box. You can either select a string from the drop down list of strings used in previous searches or enter a new string.

Match case:

Check the "Match case" box if you only want to find strings that exactly match the case of the characters in the "Find what" string.

Result Pane

The Result pane lists the Find results that have not been skipped (using the Skip button). You can double-click on the position of a result item to see the text in the PowerCOBOL Editor or use the pop-up menu described below.

The results are listed with:

Position:

The location of the string in the project. Includes starting and ending row and column numbers.

Text:

Displays the contents of the line containing the "Find what" string.

Results Pane Pop-up Menu

If you right click on the position of a result item Find displays a pop-up menu with the following functions:

Edit:

Displays the procedure containing the selected line in the PowerCOBOL Editor with the string highlighted.

Cut:

Copies the string to the clipboard and deletes it from the selected location in the procedure.

Copy:

Copies the string to the clipboard.

Delete:

Deletes the string from the selected location in the procedure.

NOTE

These functions only work if you are using the PowerCOBOL Editor.

CAUTION

Updates to procedures after a Find operation has been executed may invalidate the Find result positions so be sure you have the correct text selected before using these operations.

Replacing Text

The Project Manager Replace (All) function searches for the specified string throughout all event and COBOL procedures belonging to the selected part of the project tree, and replaces it with the replacement string. You can replace all occurrences or have the Replace function step you through each occurrence of the specified string.

The Replace dialog is shown and explained below:

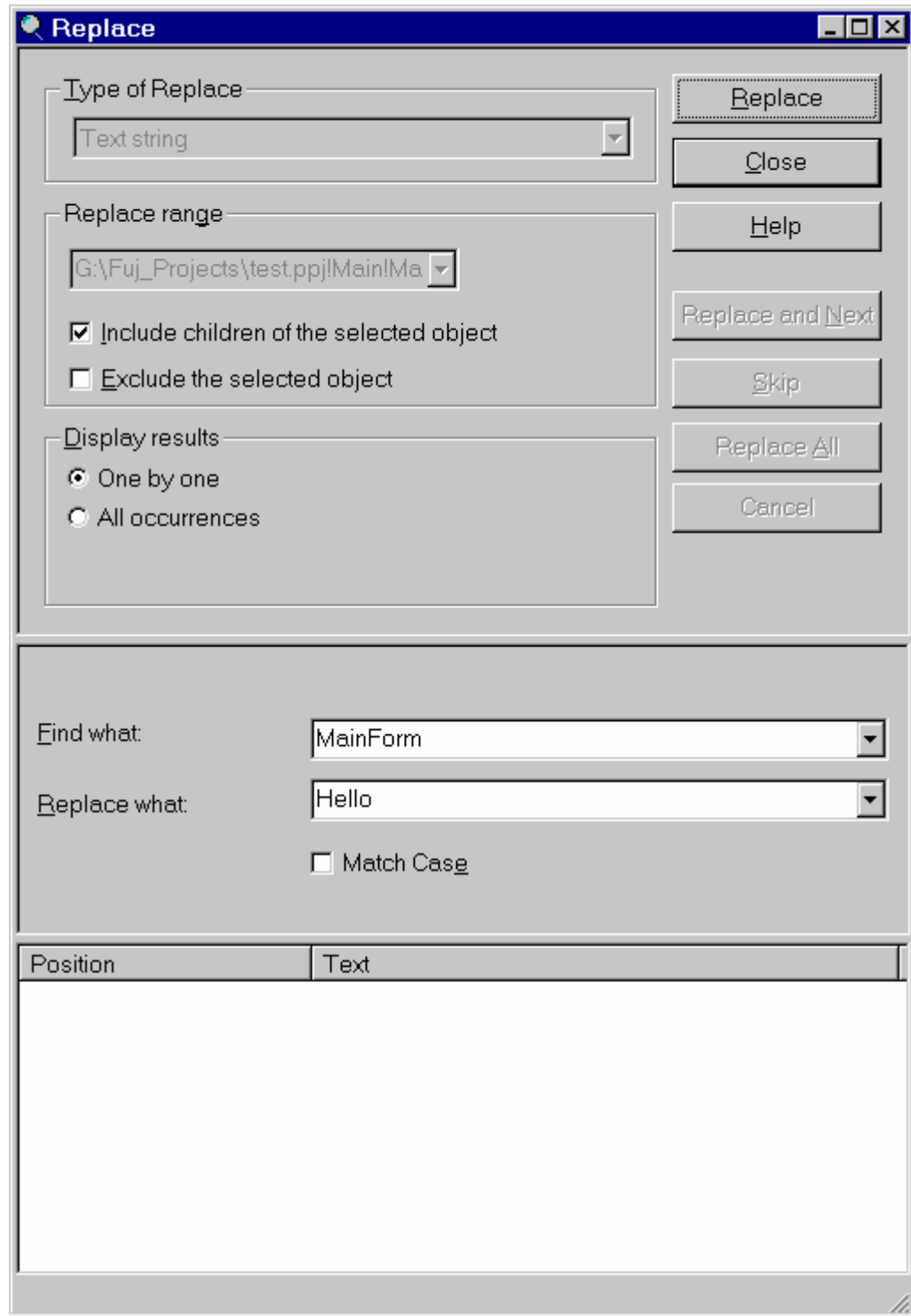


Figure 5.16. The Replace dialog window

The Replace dialog is made up of three panes: the Control pane, the Target pane and the Result pane. The contents of these panes are described below.

Control Pane

The control pane provides the functions for specifying the type and range of the Replace and for controlling the Replace operation. You specify the replace range and method of handling the results before you start the Replace operation.

Type of Replace:

Currently there is only one type of replace supported – replacing text – so the selection is disabled. This will be expanded in future releases.

Replace Range:

The drop down list shows the name of the project being searched. This field is disabled.

Includes children of the selected object:

Check this box to have Replace search objects that are on branches of the selected object in the project tree.

Excludes the selected object:

Select this box if you only want to search objects that are on branches of the selected object and not the selected object. The check box is disabled if “Includes children of the selected object” is not selected.

Display results – One by one:

Select this button if you want Replace to stop each time it makes a find.

Display results – All occurrences:

Select this button if you want Replace to search through the whole range, replacing all matching strings, and displaying results in the Result pane.

Replace Button:

The Replace button initiates the Replace operation, locating the first string that matches the “Find what” string. The Replace operation continues until all matching strings have been found or until the Cancel button is pushed.

Close Button:

The Close button closes the Replace dialog. It is disabled during a Replace operation.

Replace and Next Button:

The Replace and Next button is enabled when a Replace one-by-one operation starts. It replaces the currently located string with the “Replace what” string, registers the result of the Replace in the Result pane and searches for the next occurrence of the “Find what” string.

Skip Button:

The Skip button is enabled when a Replace one-by-one operation starts. It does NOT replace the currently located string, does NOT register anything in the Result pane, and searches for the next occurrence of the “Find what” string.

Replace All Button:

The Replace All button is enabled when a Replace one-by-one operation starts. It causes the Replace operation to search for all remaining occurrences of the “Find what” string, replace those strings with the “Replace what” string, and display the results in the Result pane.

Cancel Button:

The Cancel button is enabled when a Replace one-by-one operation starts. It terminates the Replace operation so that you can start a new Replace operation or close the Replace dialog.

Target Pane**Find what:**

Specify the string for which Replace should search in the "Find what" combo box. You can either select a string from the drop down list of strings used in previous searches or enter a new string.

Replace what:

Specify the string that Replace should use to replace "Find what" strings. You can either select a string from the drop down list of strings used in previous replaces or enter a new string.

Match case:

Check the "Match case" box if you only want to find strings that exactly match the case of the characters in the "Find what" string.

Result Pane

The Result pane lists the locations where strings have been replaced. You can double-click on the position of a result item to see the changed text in the PowerCOBOL Editor or use the pop-up menu described below.

The results are listed with:

Position:

The location of the replaced string in the project. Includes starting and ending row and column numbers.

Text:

Displays the contents of the line containing the "Find what" string, before the string is replaced.

Results Pane Pop-up Menu

If you right click on the position of a result item Replace displays a pop-up menu with the following functions:

Edit:

Displays the procedure containing the selected line in the PowerCOBOL Editor with the location of the original string highlighted.

Cut:

Copies the characters that now occupy the location of the original string to the clipboard and deletes those characters from the selected location in the procedure.

Copy:

Copies the characters that now occupy the location of the original string to the clipboard.

Delete:

Deletes the characters that now occupy the location of the original string from the selected location in the procedure.

NOTE

These functions only work if you are using the PowerCOBOL Editor.

CAUTION

Updates to procedures after a Replace operation has been executed may invalidate the Replace result positions so be sure you have the correct text selected before using these operations.

Chapter 6. Creating Executable Programs

This chapter describes how to use the PowerCOBOL compiler to create an executable program including:

- Compiling and linking projects
- Creating a project
- Building a project
- Batch Building
- Installing Applications (creating custom setup programs for your applications)
- Using Precompilers with PowerCOBOL

Overview

PowerCOBOL manages applications as *projects*. A project contains all of the resources required by the application including the application windows (forms), event procedures, bit maps, icon files, point (cursor) files, imagelist files, COBOL source files and library files.

Projects are organized by application modules. Modules are organized by the objects contained within them such as forms (windows). Lastly, forms are organized by objects within them such as controls, events, methods, properties and event procedures (COBOL "scriptlets").

Each resulting module executable may either be an .EXE file or a .DLL file. PowerCOBOL .DLL files can be shared by any application, including other language executables such as Visual Basic and C++.

The steps you take to create an executable program are as follows:

1. Create a new project in the PowerCOBOL Project Manager.
 - a. Select New Project from the File menu.
 - b. Select the appropriate application template (for example, "Standard Form").
2. Expand the project by right clicking the mouse on the project name in the left windowpane and selecting Expand All from the pop-up menu. You will see that a default form has already been created for you. This will be the starting form for the application.
3. If you wish to add additional forms to an existing application, right-click the mouse on the module name in the left windowpane and select Create Form from the pop-up menu.
4. Once you have one or more forms created, edit each form by right clicking the mouse on the form name and selecting Open from the pop-up menu. You then add the controls you wish and write any needed event procedures.
5. When you are finished editing your forms, save them and save the project definition by selecting Save from the File menu in the PowerCOBOL Project Manager.

6. You are now ready to compile and link the project into an application. You do this by selecting Build All from the Project menu. You may alternatively select the Build ALL option from a pop-up menu by right clicking the mouse on the project name in the left windowpane.

NOTE: This version of PowerCOBOL can import projects from previous versions of PowerCOBOL, including version 1 and up. Select Open from the File menu and then select "PowerCOBOL V3.0 or earlier project (*.prj)" from the Files of Type File Type option if you wish to open a project file from V3.0 or earlier version of PowerCOBOL. Select "PowerCOBOL Project (*.ppj)" if you wish to open a V4.0 or higher project.

Building an application invokes the PowerCOBOL compiler upon the module. If the compile is successful, the PowerCOBOL linker will then be invoked to create an executable module.

Compiling translates any high-level symbolic descriptions into a lower-level symbolic or machine-readable format. Linking gathers object modules and related resources, merges them, and resolves reference conflicts to produce an .EXE program.

When the Build command is issued, PowerCOBOL checks the update status to compare the modification dates of source files. PowerCOBOL then *automatically* selects the required modules to recompile, and re-links any files that have a modification date later than the resultant object module.

A Rebuild All, on the other hand, is the recompile of *all* source files and a re-link of *all* object modules regardless of modification dates.

Project Files

You create an executable program by 'building' the project. The following files can be held in a project:

Project Definition file:	with a .PPJ file extension
COBOL source:	with a .COB file extension
Icon:	with a .ICO file extension
Bitmap:	with a .BMP or .DIB file extension
Cursor:	with a .CUR file extension
ImageList:	with a .BMP file extension
Library file:	with a .LIB file extension
Object file:	with a .OBJ file extension

Adding Forms (Windows) to a Project Module

To add forms to a project module, expand the project in the PowerCOBOL Project Manager left windowpane. Move the mouse over the module name and right-click to display a pop-up menu. Select Create Form. A new form will be added.

If a module contains multiple forms, you may wish to change the starting form (the first form that is displayed when the application is started). You do this in the left windowpane of the PowerCOBOL Project Manager, by right clicking the mouse on the name of the form you wish to be the starting form. From the pop-up menu, select Property to display the Properties Dialog box as follows:

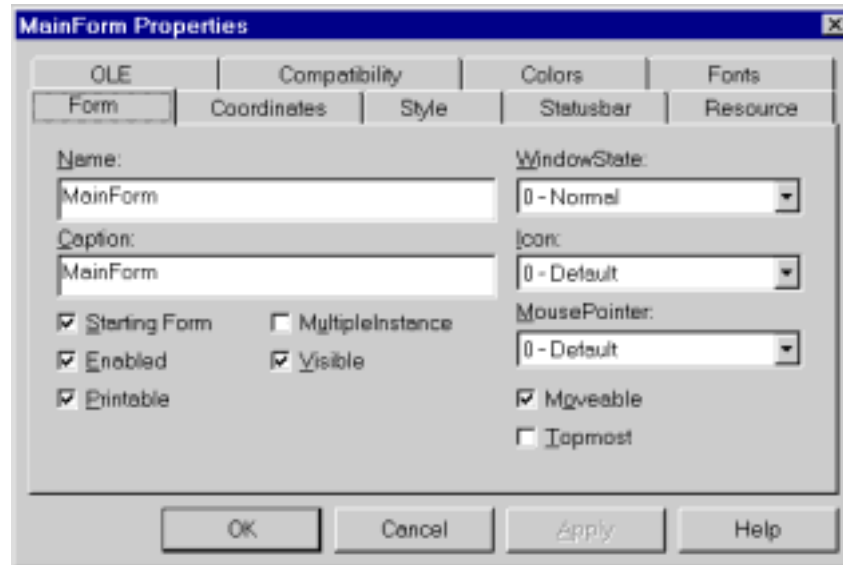


Figure 6.1. The form properties dialog box

Select the Starting Form check box to make the form the starting form for the module that contains it.

Adding Objects to a Form

To add objects to a form, open the form for editing by right clicking on the form name and selecting Open from the pop-up menu.

The form will be opened in the PowerCOBOL Form Editor as shown in the following figure:

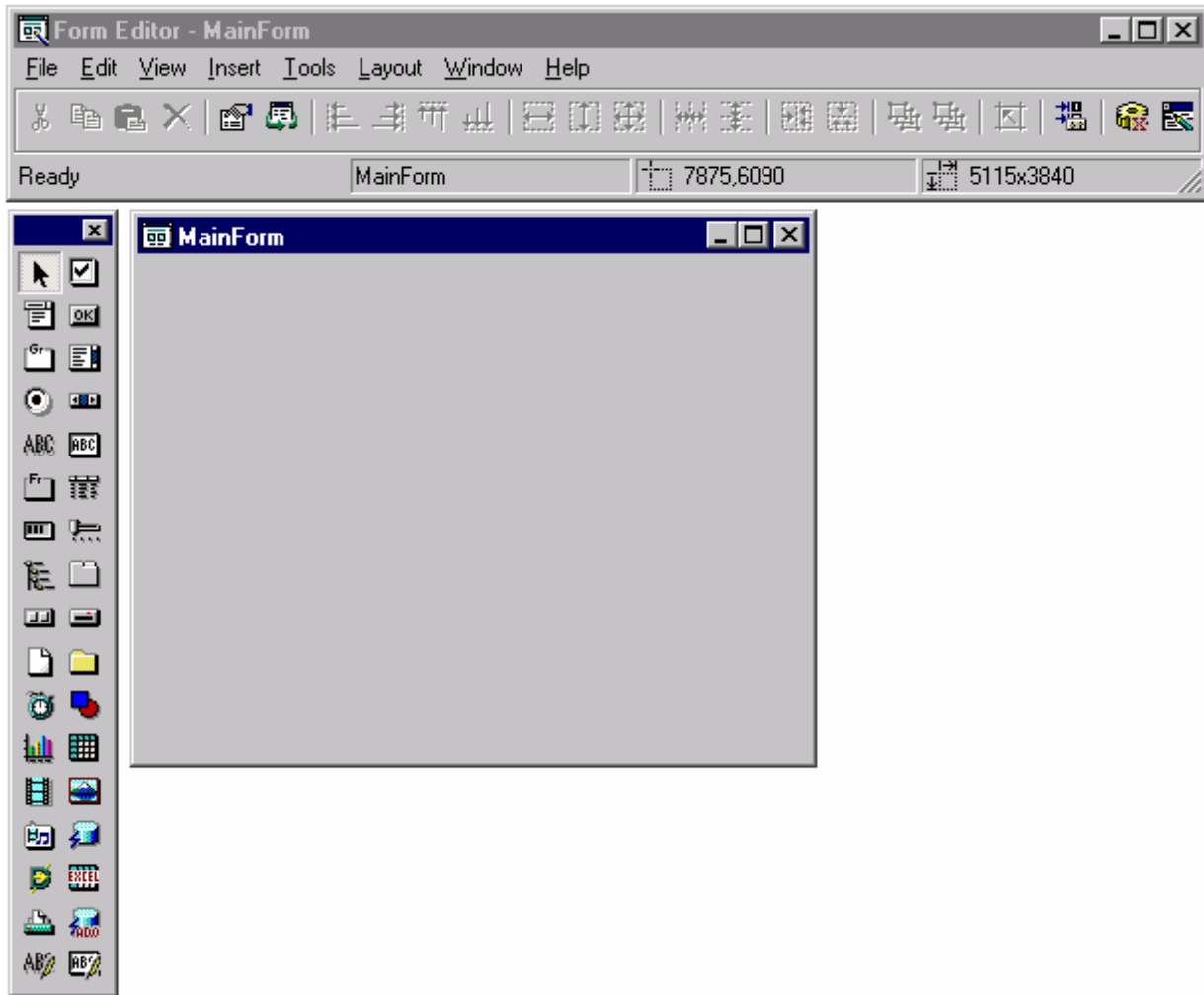


Figure 6.2. The PowerCOBOL Form Editor

Objects may be added to a form in one of three ways:

1. By selecting an object from the Toolbox palette and dropping it onto the form. Alternatively, you can select a control by selecting the Insert Control option from the Insert menu.
2. You may select some objects such as icons from a form's Properties dialog box (under the Resources tab) by right clicking the mouse on the form and selecting Properties from the pop-up menu.
3. You may select custom controls (such as ActiveX, OCX's and OLE and COM objects) available by selecting Custom Controls from the Tools menu. This

displays the Custom Controls dialog box that shows the custom control groups available and allows you to browse through the control groups as follows:

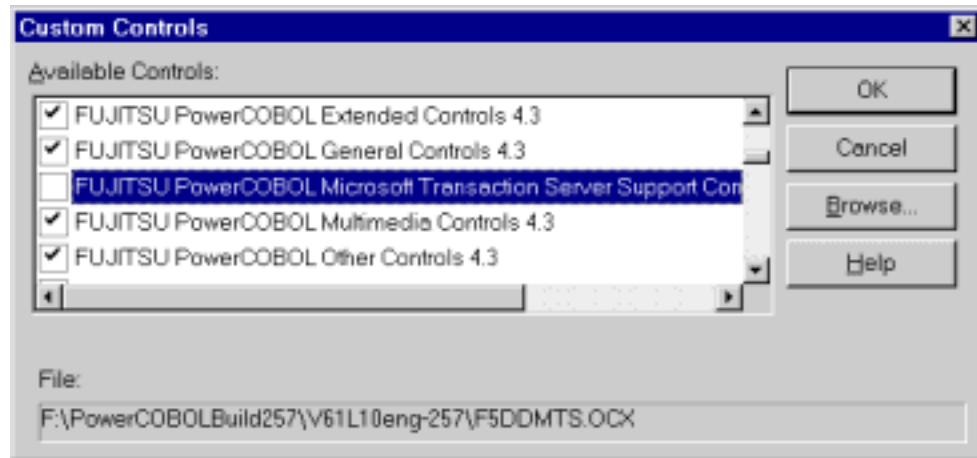


Figure 6.4. The Custom Controls dialog box

You may select any one of these control groups and click the Browse button to see a list of individual custom controls available in that group.

Adding Modules to a Project

When you initially create a project you will have a default main executable module defined automatically. This single module may contain multiple forms and thus encapsulate the entire application into a single executable file.

If you wish to create additional modules within an application, you may do so by right clicking the mouse on the project name in the left windowpane of the PowerCOBOL Project Manager and selecting the Create Module option from the pop-up menu.

You may then create forms and other related objects under a new module. The module type will default to an Execute Module (.EXE file). You may change this in the module's properties to be a .DLL file if needed.

Opening an Existing Project

You may open an existing project in the PowerCOBOL Project Manager by selecting Open from the File menu. Use the dialog box to navigate to the appropriate location and then select the project file you wish to open.

Building an Individual Module

You may compile and link an individual module in a project by right clicking on the module name in the left window pane of the PowerCOBOL Project Manager and selecting Build from the pop-up menu.

This will cause all of the module's related application components to be re-compiled and the module and its components will be linked into a single executable file. If an error occurs during the compilation phase, the link will be aborted.

If you select the Build option instead, only those application components that have changed since the last build will be re-compiled, and the application will be re-linked into a single executable. If an error occurs during the compilation phase, the link will be aborted.

Module properties

If you right click on a module name and select Properties from the context menu that appears, you will be presented with the Module Properties dialog window as follows:

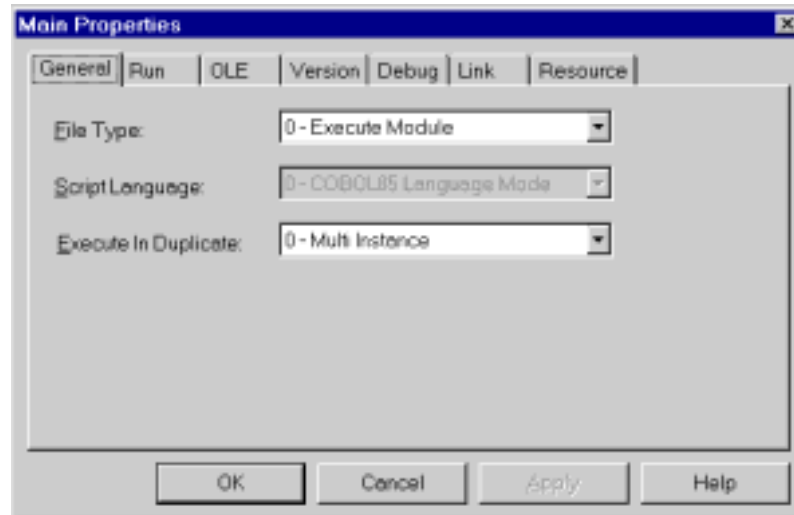


Figure 6.5. Module Properties dialog window

FileType

Specifies the target file type - execute module (.EXE) or dll module (.DLL).

If you change the type to DLL module without using "ActiveX" template project and create the module as a custom control, note that:

- The custom control is created with the OOCOBOL language mode. (You cannot change the mode.)
- If you define EXTERNAL data/file in a custom control and it's container (user of the custom control) uses another data/file, which has same name as the one of the EXTERNAL data/file in the custom control, the data/file area may be invalid or the application may abend at execution time. Therefore, it may not be a good idea to use the EXTERNAL data/file in a custom control, which is provided to the public.
- With the COBOL85 language mode, you cannot set the MultipleInstance property of form to true. So, the custom control made in this type cannot be used multiple in a same process.

Script Language

If you select "1 - OOCOBOL Language mode" PowerCOBOL can use the OOCOBOL programming features. You can change the mode only when any Script has not been created in the module. Refer to "COBOL Language Reference" for details of OOCOBOL.

If you are going to make an ActiveX control that may be used in multiple instances, you should set the Language mode to "OOOBOL Language mode".

Execute In Duplicate

Specify whether to execute the module in duplicate.

Beginning in version 6.1, there are three separate behaviors available:

- 0 -Multi Instance - allows multiple instances of the application.
- 1 - Single Instance (Message) - allows only one instance of the application and displays an error message if the user attempts to execute a second instance.
- 2 - Single Instance (Activate) - allows only one instances and focuses the user on the first instance of the application already running if he/she attempt to launch a second instance.

The Module Properties Version settings

If you click on the version tab of the Module Properties dialog window, you will be presented with the version settings as follows:

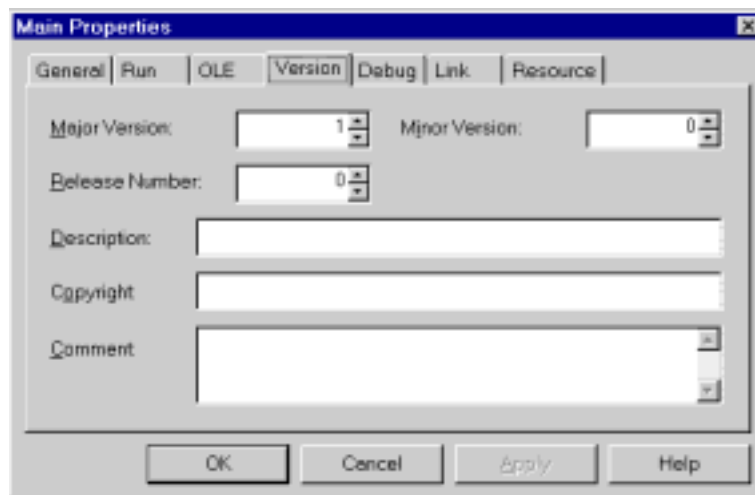


Figure 6.6. The Module Properties Version tab

Major Version and Minor Version

Specifies the version of the execution file (exe/dll) made from the module. If you create an ActiveX control, you should specify this information. It will be registered to the system when you register the module.

Release Number

Specifies the release number of the execution file made from the module. This is not registered to the system.

Description

Specifies the description of the execution file made from the module. If you create an ActiveX control, you should specify this information. It will be registered to the system when you register the module.

Copyright

Specifies copyright information.

Comment

Specifies comments regarding the execution file.

Building an Entire Project

You can build an entire project, including all application modules, in a single step in PowerCOBOL.

You do this by right clicking the mouse on the project name in the left windowpane of the PowerCOBOL Project Manager Window, and selecting Build All from the pop-up menu.

This will cause each module in the application and its related components to be compiled and linked into separate executable files. If an error occurs during the compilation or link phase of an individual module, the remainder of the build process will be aborted.

If you select Build All option, only those application modules and their components that have changed since the last build will be re-compiled, and only the changed modules will be re-linked into new executables. If an error occurs during the compilation or link phase of any individual module, the remainder of the rebuild process will be aborted.

Batch Building

PowerCOBOL can build/rebuild all modules included in a project by using batch (DOS) mode. The result of the build/rebuild is output into the ".blg" file in the folder where the project file exists.

Command Syntax Format

The following format is the command statement used for batch building:

```
PowerCOB { /build | /rebuild }  
[ /Debug | /Release ]  
[ /cbi:"OPTION-FILE-NAME" ]  
"PROJECT-FILE-NAME"
```

The components of this command statement are:

/build | /rebuild

Specifies whether to build or rebuild a module included in the project file.

/Debug | /Release

Specifies whether to build the module in debug mode or release mode. If the mode is omitted, PowerCOBOL uses the mode specified in the module's properties.

/cbi:"OPTION-FILE-NAME"

Specifies the file that contains compile options. You can create this option file by using the "WINCOB" command. See "How to create an option file" for details. If an option file is specified, the options (both copy folders and copy file names) specified

in the script properties are ignored. If the option file is omitted, PowerCOBOL uses the options specified in the script properties.

"PROJECT-FILE-NAME"

Specifies the project file name you wish to build. Note that project files from V3.0 or earlier cannot be specified.

Returning Value

Succeeded: 0

Failed: 1

"Succeeded" implies I-level and W-level errors only. If the project has already been built, the batch building succeeds. See the result file ".blg" for details.

Example

If you would like to build a project named "c:\project\proj1.ppj":

```
powercob /build "c:\project\proj1.ppj"
```

The result will be output as follows in the "c:\project\proj1.blg".

```
Create Type Library for compilation...
Create C:\project\Main\Debug\MainForm.cob...
STATISTICS: HIGHEST SEVERITY CODE=I
Revise line information into C:\project\Main\Debug\MainForm.cob...
STATISTICS: HIGHEST SEVERITY CODE=I
Compile C:\project\Main\Debug\MainForm.cob...
STATISTICS: HIGHEST SEVERITY CODE=I, PROGRAM UNIT=1
Compile resource C:\project\Main\Debug\Main.rc...
Linking C:\project\Debug\Main.exe...
** The build was successful **
```

If you'd like to build a project named "c:\project\proj2.ppj" specifying with an option file and debug mode:

```
powercob /build /debug /cbi: "c:\project\proj2.cbi"
"c:\project\proj2.ppj"
```

If there are any compile errors, the results are output as follows in the "c:\project\proj2.blg" file.

```
Create Type Library for compilation...
Create C:\project\Sub\Debug\SubForm1.cob...
STATISTICS: HIGHEST SEVERITY CODE=I
Revise line information into C:\project\Sub\Debug\SubForm1.cob...
STATISTICS: HIGHEST SEVERITY CODE=I
Compile C:\project\Sub\Debug\SubForm1.cob...
** DIAGNOSTIC MESSAGE ** (SUBFORM1)
SubForm1 SubForm1-Opened(5) : JMN2503I-S USER WORD 'DATA1' IS UNDEFINED.
SubForm1 SubForm1-Opened(5) : JMN2557I-S FORMAT OF DISPLAY STATEMENT IS
INCOMPLETE.
STATISTICS: HIGHEST SEVERITY CODE=S, PROGRAM UNIT=1
** The build has failed **
```

If you'd like to build a multiple number of projects, create a batch file and execute the command as follows:

Batch command

```
allbuild rebuild c:\temp\allbuild.blg
```

Contents of the batch file "allbuild.bat"

```
ECHO OFF

ECHO ##### C:\project\proj1.ppj ##### >> %2
ECHO POWERCOB /%1 "C:\project\proj1.ppj"
START /WAIT POWERCOB /%1 "C:\project\proj1.ppj"
IF ERRORLEVEL 1 ECHO !!! %1 Error !!!
TYPE C:\project\proj1.blg >> %2

ECHO ##### C:\project\proj2.ppj ##### >> %2
ECHO POWERCOB /%1 "C:\project\proj2.ppj"
START /WAIT POWERCOB /%1 "C:\project\proj2.ppj"
IF ERRORLEVEL 1 ECHO !!! %1 Error !!!
TYPE C:\project\proj2.blg >> %2

...

:END
```

How to create an option file

You can create a compile option file by using the "WINCOB" command. Execute the command and then display the compile option dialog box to specify compile directives.

```
WINCOB -iOPTIONFILE-NAME
```

For example, if you create an option file named "C:\project\proj2.cbi", specify as follows:

```
WINCOB -iC:\project\proj2.cbi
```

Refer to the COBOL97 User's Guide for details of the WINCOB command.

NOTES

- Command statements used in batch building are not case sensitive.
- Use the "LIB" option for compiling when you specify any copy files using the /cbi option.
- If you move a project file to another folder and build the project, PowerCOBOL outputs the message "Would you like to save the project for build or compilation?". You will not be able to use the batch building feature as a result. In this case, check the "Auto Save" in the Build tab of the Option properties dialog. The Option properties dialog is displayed when you select the "Options" option in the "Tools" menu of the Project window.

Installing Applications

Registering and Deleting

If you want to use a form as an OCX control, you need to register it to the system. To register a form as an OCX control:

1. Build the module to create a DLL without errors.
2. Select the module in the left windowpane of the PowerCOBOL Project Manager.
3. Either select Register from the pop-up menu or, from the Edit menu select Object, Register.

If the OCX control is no longer required, you can unregister it by:

1. Select the module in the left windowpane of the PowerCOBOL Project Manager
2. Either select Unregister from the pop-up menu or, from the Edit menu select Object, Unregister.

Note that when you unregister a control, project files that use it can be opened but cannot be updated.

When you use the OCX control on another computer, use an installer to register the control on the computer. Similarly, when the control is no longer required, unregister it with an uninstaller.

Creating an Installer

PowerCOBOL can create an installer for your PowerCOBOL application files. To have PowerCOBOL create the installer, select "Make Installer" from the File menu. PowerCOBOL creates the following files in the output folder (..\debug or ..\release depending on the project's BuildMode property):

- setup.exe, setup.inf - For installing
- uninst.exe - For uninstalling
- f5ddstev.exe - For setting the environment

When you run setup it prompts the user for an installation folder, creates it if necessary and installs the PowerCOBOL application files to that folder. It also installs files uninst.exe and uninst.inf used to uninstall the application.

When you uninstall an application, use "Add/Remove Programs" in the Windows Control Panel. If you execute uninst.exe directly, you cannot uninstall the application correctly.

Executing f5ddstev.exe determines whether a message box displayed when an automation error occurs at execution time. This setting works for all PowerCOBOL applications running on the computer. F5ddstev.exe is executed from the command line using one of these formats:

Display the error message:	f5ddstev /ERRMSGBOX:SHOW
Do not display the error message:	f5ddstev /ERRMSGBOX:HIDE

Installing Complete Applications

NetCOBOL provides an "installation wrapper" that helps you install all parts of your NetCOBOL application, including the PowerCOBOL installer and associated files. See the "NetCOBOL Runtime Installation Guide" for full details.

Using Precompilers with PowerCOBOL

If you wish to use statements, like EXEC SQL, that have to be precompiled, you can configure PowerCOBOL to invoke the precompiler for both event procedures and external COBOL files included in the PowerCOBOL project.

This section explains how to configure PowerCOBOL so that it will invoke a precompiler when compiling COBOL code.

Configuring PowerCOBOL with a Precompiler

You tell PowerCOBOL to use a precompiler by editing the Precompiler properties for each COBOL Script group and each external COBOL file that needs the precompiler.

To access the Precompiler Properties dialog:

1. In the PowerCOBOL Project Manager, right click on a COBOL Script node or a COBOL File node in the project tree.
2. From the pop-up menu select Properties.
3. Select the Compile tab.
4. Press the Precompiler Set button.

PowerCOBOL displays the Precompiler Properties dialog, shown below:

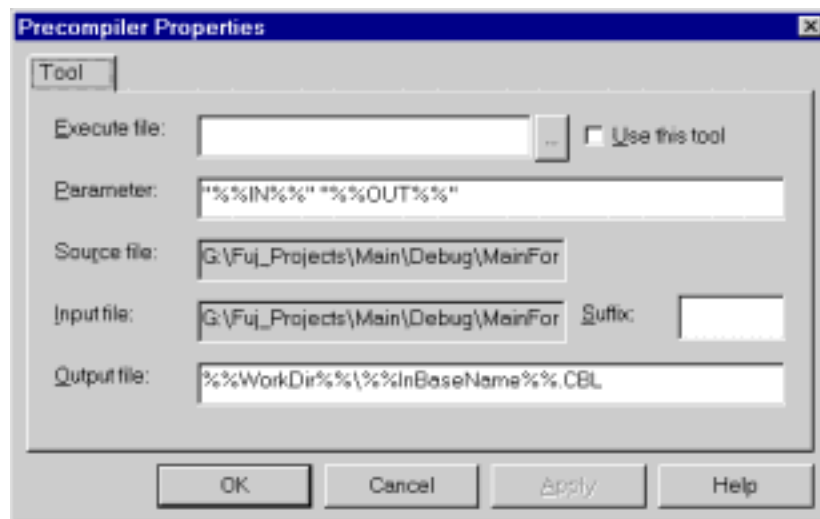


Figure 6.7. Precompiler Properties dialog window

Where:

Execute file:

Specifies the name of the precompiler executable.
It can be an .EXE file or a .BAT file.

Use this tool:

Specifies whether to use the precompiler set in the Execute file field. A preprocessor is executed only when this is checked.

Parameter:

Specifies the parameters to pass to the precompiler. You can include macro strings described in "Macros" below.

For example, if the command line of the precompiler is:

"Execute-file-name -I Input-file -O Output-file"

you would enter the following in the parameter field:

-I %%IN%% -O %%OUT%%

Source file:

For event procedures contains the name of the COBOL source file that is created when the project is built.

For external COBOL programs contains the name of the COBOL source file.

Input file:

Displays the name of the file that will be input to the precompiler.

If you do not enter anything in the Suffix field (below), this is the same as the Source file.

If you specify a Suffix, then the Input file name is made by using the Output file name with its extension changed to that specified in the Suffix field.

Suffix:

Use this field to change the extension of the input file for the precompiler. This field should be used when the precompiler only accepts a fixed extension for the input file and that extension is different from the Source file extension.

Output file:

Specifies the name of the file to be output from the precompiler. This can be passed to the precompiler by using the %%OUT%% macro in the Parameter field.

Macros

You can use any of the following macro strings in the Parameter field of the Precompiler Properties dialog.

%%IN%%

Is replaced by the name in the Input file field.

%%InBaseName%%

Is replaced by the name in the Source file field with the extension removed.

%%InDir%%

Is replaced by the path name of the Source file.

%%InSuffix%%

Is replaced by the extension of the Source file.

%%OUT%%

Is replaced by the name in the Output file field.

%%OutDir%%

Is replaced by the path name of the Output file.

%%ProjectDir%%

Is replaced by the path name of the project file (.ppj).

%%WorkDir%%

Is replaced by the path name of the folder used for debugging and building. When PowerCOBOL builds projects it creates a subfolder to the folder containing the project for each module in the project, using the name of the module. Within that folder it creates either a "debug" or "release" folder depending on the setting of the project's BuildMode property. %%WorkDir%% returns the name of this folder.

Examples of Configuring Precompilers**Oracle Pro*COBOL**

To use the Oracle Pro*COBOL precompiler for Windows 95/NT, version 1.8 for external COBOL files make the following settings:

Execute file: procob.exe
Parameter: %%IN%% %%OUT%%
Output file: %%WorkDir%%\%%InBaseName%%.CBL
Use this tool: Check the checkbox

SymfoWARE Espl-COBOL

(This product is not available in the US market)

To use the "Espl-COBOL" of "SymfoWARE Programmers Kit" for Windows NT, version 1.1L10 for PowerCOBOL event procedures make the following settings:

Execute file: splpcob.exe
Parameter: -t %%WorkDir%% %%IN%%
Suffix : sco
Output file: %%WorkDir%%\%%InBaseName%%New.cob

NOTE

You cannot use ".COB" as an extension of an input file to the Espl-COBOL precompiler. Therefore you need to specify an extension in the Suffix field.

In the Output file field, the name specified in the Input file field with ".COB" extension is always displayed. Therefore you need to use the extension ".COB" when you specify the Output file. In this example, "New" is added to the file names to prevent them from overlapping with the source file name.

Chapter 7. Debugging the Program

This chapter explains how to debug an executable program using PowerCOBOL including:

- Using the PowerCOBOL Debugger in an event-driven environment
- The PowerCOBOL debugging environment
- Setting, disabling and removing breakpoints
- Monitoring and changing data values
- Controlling application execution
- Diagnostic Reporting

Overview

The PowerCOBOL Debugger helps you find, identify and fix errors in your application. It allows you to:

- control program execution
- set breakpoints
- monitor and dynamically change data

You can instruct a program to execute in different ways: unconditionally, by stepping over entire procedures, or in single statement steps.

Unconditional execution runs straight through an application, pausing only for breakpoints and user interface input. You may, for example, set a breakpoint at a particular execution location, and run the application through in high speed to get to that point, without being forced to single step through every prior execution statement.

Stepping over entire procedures (executing an entire procedure or called program as a single step) allows you to exercise some control in stepping through the logic of an application without being forced to watch every individual logic branch and execution statement while debugging.

Single step execution stops the application after each instruction, thus allowing you to watch every individual execution statement, with the option of examining and changing data and object properties at any time.

A *breakpoint* is a marker that you put into one or more locations in an application to force the Debugger to pause, and return control to you at a specified location in your source code before proceeding. The PowerCOBOL Debugger supports both conditional and unconditional breakpoints.

An unconditional breakpoint forces the execution to pause at a given execution statement anytime the flow of execution encounters the statement. Unconditional breakpoints can be set at any execution point within a program.

A conditional breakpoint will only pause execution at a given execution statement if a specific condition you've specified has been met (for example, a data item named "My-Counter" being equal to 10). Conditional breakpoints save you from having to

step through every iteration of a specific statement and having to manually check a data item's value each time.

The PowerCOBOL Debugger also allows you set conditional breakpoints based upon whenever a specific application *event* occurs (for example, when a user clicks on the OK button), or whenever a specific program in the application is entered, or even when a specific function within a specific program in an application is invoked.

You can additionally monitor any data item in the application through the use of the Watch and Quick Watch options. The *Watch* option will display the data item name and its current value in a list within the main Debugger window. This allows you to monitor a data item during execution. The Quick Watch option will show you the current value of a data item without adding it to the list of data items being monitored in the main Debugger window. Additionally, Quick Watch will allow you to change a data item's current value dynamically ("on the fly").

The PowerCOBOL Debugger also allows you to make program modifications to the source code as you debug. These modifications will not take effect until you re-build the application. This allows you to debug, make changes, and continue testing within the same highly integrated PowerCOBOL development environment.

When you find an error in a program, you may immediately correct the problem. After removing the error, you have the option of continuing the current test with the Debugger or of saving your changes, rebuilding the application and restarting the debug process. Continue with either process until all errors have been removed from the program.

Debugging in an Event-driven Environment

The PowerCOBOL Debugger has been carefully designed and integrated to provide optimal usage in an event-driven environment.

PowerCOBOL itself sits on top of a highly complex graphical user interface (GUI) based system.

In this GUI-based system, hundreds and possibly thousands of events may occur within a few seconds. These events may include something as specific as a user clicking on a close button or something as simple as a user moving a mouse over a particular portion of an application window with no specific intent.

The Windows operating system manages each application process and the GUI display separately. When a GUI event takes place, Windows determines which application(s) are related and will compose and dispatch messages to this application(s).

For example, you may have noticed that many Windows applications (including PowerCOBOL's development environment) watch closely for a user to move the mouse pointer over an icon in the menu area. When the mouse pointer moves over any such icon, a small text box appears describing what function that icon will accomplish if selected (left clicked on).

In order for the actual application that comprises the PowerCOBOL development environment's user interface to provide this useful feature, it must first be told just when a user moves the mouse over a particular icon.

The Windows operating system accomplishes this task by recognizing when such an *event* takes place, composing a message saying so - including all needed details, and dispatching (sending) it into the appropriate application.

It is important to understand that the Windows operating system will compose and dispatch all event-related messages into an application - even if the application has no interest in a particular event or series of events. In some cases an application may *ABEND* (abnormally end) because it has not had an appropriate message handler defined.

Many development environments require the programmer to be aware of this fact and to write default message handlers for many types of events they do not really care about within an application. Some other language environments provide default message handlers in their run-time systems to handle this processing, but will pass messages through if it is determined the application really does care about a certain event.

PowerCOBOL falls into the later category.

Programs running under Windows may have many thousands of messages dispatched into them while they are active.

In many lower-level programming environments such as C and C++, the application itself must handle all messages. These applications are typically constructed to contain a *message loop* - a section in the program where each message is read from the message queue and interrogated to see if the application cares about it. If the particular message is of interest, then a specific function is called to deal with it. Once the function has completed its task, control is returned to the message loop and the next message is read and interrogated.

When using a Debugger in such an environment, you must be very careful to set breakpoints in appropriate places. If you have the misfortune of stepping into the actual message loop code, you may never get out, as the messages pile up very quickly, and you may generate numerous additional messages as a function of debugging the message loop. Because so many messages stack up so quickly, additional commands to the Debugger may not get through until the previous messages have been dealt with.

Another issue, when programming in a lower-level environment, is that during debugging you can easily step into source code that you did not write. There is a great deal of run-time support code and GUI abstraction code provided by any programming tool in a Windows GUI environment.

This code is typically linked into these applications from libraries provided with the development environment.

One of the features of PowerCOBOL that you will appreciate during debugging is the fact that as PowerCOBOL abstracts a great deal of the programming task, you never have to look at this run-time support code when debugging. You typically only debug code that you write.

This greatly simplifies the overall development process, and generally shortens the debugging task.

The PowerCOBOL Debugger is aided by the fact that the design of the PowerCOBOL system itself attempts to abstract much of this work from the application developer. Developers using PowerCOBOL only define event procedures for events they are concerned about within the application. The PowerCOBOL run-time environment takes care of the multitude of other Windows- composed event messages sent into a PowerCOBOL application.

This greatly simplifies both the application design and the debugging task.

It is important to realize, however, that a great deal of application processing is thus abstracted and is taking place transparently along with the processing you do see when debugging.

This may cause performance problems with the Debugger. It may, at times, appear slow when it is dealing with a large number of extraneous messages, or waiting for Windows to handle other processes.

When using the PowerCOBOL Debugger with larger and more complex applications (as is true with any Debugger in the Windows environment), you may at times want to pause slightly between mouse clicks or keyboard commands, or you might have such a request ignored (skipped). For example, you may actually lose a command because you get too far ahead of the message queue handling by the PowerCOBOL run-time system.

You may at times think the Debugger is locked up and not responding, when in fact the Debugger has returned control to your application and is simply waiting for an event to occur in the application, before returning control back to the Debugger.

Another potential problem comes from one of the very basic design decisions made in the Windows operating system.

An application may create (paint) a window on the screen that includes text and graphical objects (for example, menus and push buttons).

Because Windows is a multiprocessing operating system, thus allowing multiple applications to run concurrently using the same display, your application's window may be overlapped by another application's window being painted on the screen.

Windows does not always keep track of its processes, however; it will not necessarily "refresh" the screen automatically to show your application's current window.

In general, the Windows operating system does not keep track of everything inside of application windows. This means that if another application window overlaps your application's window on the screen, portions of your application's window that were overlapped will effectively be overwritten and lost on the display.

The effect you might see when the other application is exited and your application's window is again made visible is that all or part of your application's window will contain parts of whatever was just displayed over top of it.

Windows does not keep track of the contents of application windows because of the overhead associated with doing so. Instead, Windows will compose and dispatch a message alerting an application that one of its windows has been overlaid in such a manner, thus requesting a repaint process from the application itself. The application then repaints its window.

This is another excellent feature in the design of PowerCOBOL; it takes care of managing this situation for you and you need not worry about creating processes to repaint your application windows when this occurs.

However, it is important to understand this design feature of Windows when you are debugging in the PowerCOBOL environment. The reason behind this is that control passes between your application's GUI interface and the PowerCOBOL Debugger as you debug your application.

As previously noted, the PowerCOBOL Debugger may appear to be frozen and not responding to your requests, when in fact it has returned control to your application's GUI interface and is awaiting user interaction to generate an application event.

Likewise, at times you may see your application's window overlaid with another application's window and not refreshed. It may appear at first as if your application has some serious problem or error in it. The explanation behind this is that control may have passed from your application's GUI interface back to the Debugger. While the Debugger is in control, your application's window will remain unchanged until an event is raised within the Debugger to cause an update to the application's window. The window will then be repainted and should return to its normal look and feel.

Overall, it's important to remember that the PowerCOBOL Debugger is attempting to insert itself between the very active event-driven message architecture of the Windows operating system and your actual application. At the same time, it is trying to co-exist with and run under the auspices of the PowerCOBOL run-time system which is itself attempting to abstract a large portion of the tedious application processing from your application.

It is thus very important to understand how to best use the Debugger within your application development process. If you keep a keen emphasis on the most logical places to set breakpoints and allow the Debugger to run quickly through large parts of your application code that you are comfortable with, you will optimize your overall productivity.

The PowerCOBOL Debugging Environment

This section takes you on a tour of the PowerCOBOL debugging environment. For the purposes of this section the sample "Hello" application discussed in the first part of Chapter 3 will be referenced.

The "Hello" application consists of a single window with a simple text display control with an initial value of "Begin" and a single push button that is labeled "Hello". When the user clicks on the push button, the text string "Begin" is changed to "Hello" in the text control.

For the purposes of this chapter, the event procedure code for a user clicking on the push button (CmCommand1) has been enhanced as follows:

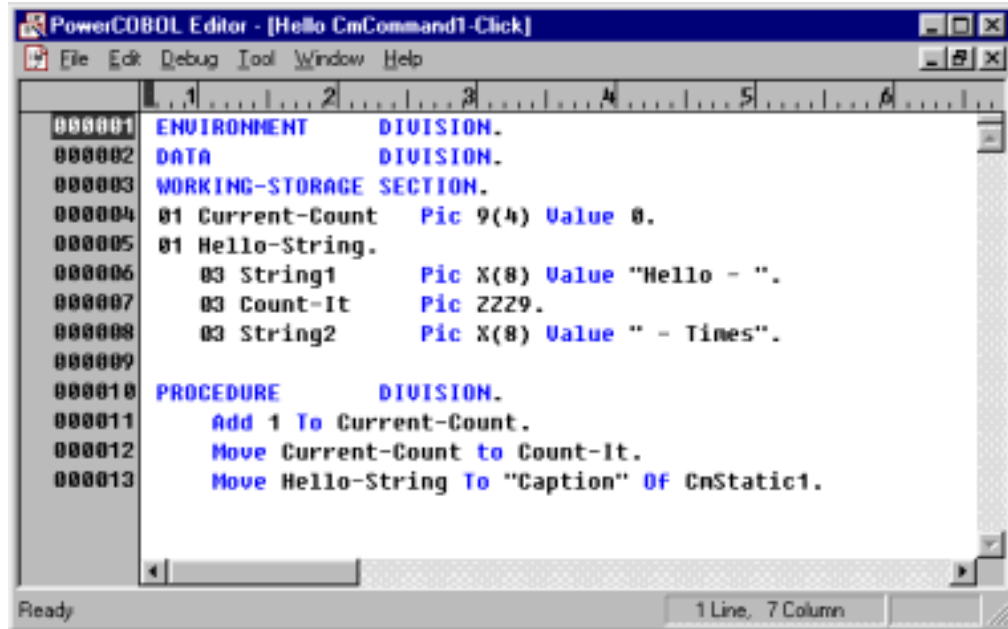


Figure 7.1. Command Button Click event procedure updated.

The above noted enhancement now keeps track of the number of times a user clicks on the Hello push button. It then displays a message in the text control of "Hello n Times" where "n" represents the number of times the user has clicked the Hello push button.

Invoking the Debugger

To use the Debugger, you must first build an application in debug mode. This is the default mode for PowerCOBOL. To ensure that your current build option setting is for debug mode, select the Properties option for the project (right-click on the project name in the left windowpane and select Properties from the pop-up menu). Within the project's Properties dialog box, click on the Build tab.

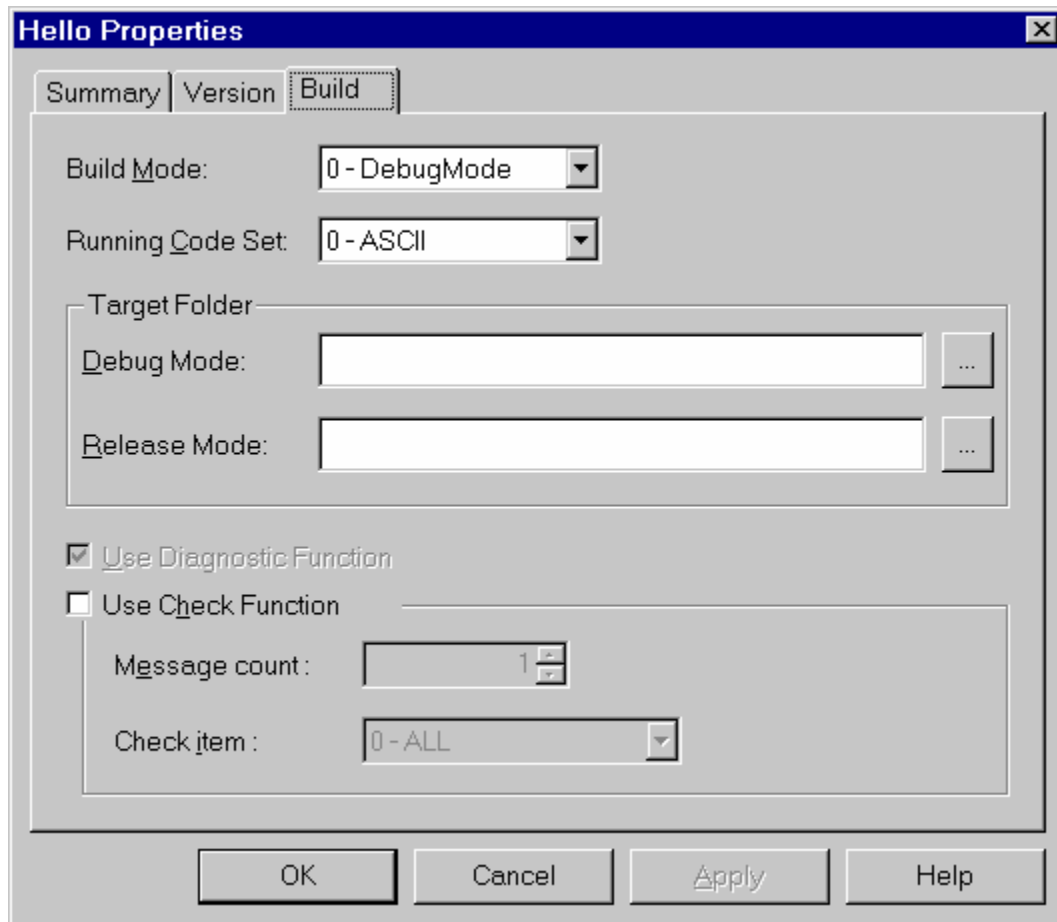


Figure 7.2. The Properties dialog box for the Hello project

Make sure that "0-DebugMode" is selected in the Build Mode field and click on the OK button. Now build the project (right-click on the project name in the left windowpane and select All Rebuild from the pop-up menu).

Once you have successfully built the project in debug mode, you are ready to invoke the Debugger.

Move the mouse over the name of the application module where you want to start the Debugger (typically the main module), right-click the mouse on it and select Debug from the pop-up menu.

The Debugger is tightly integrated into the PowerCOBOL application development facility. When the Debug window appears, right click on the project and select the Expand All option to expand your application hierarchy so that it appears as follows:

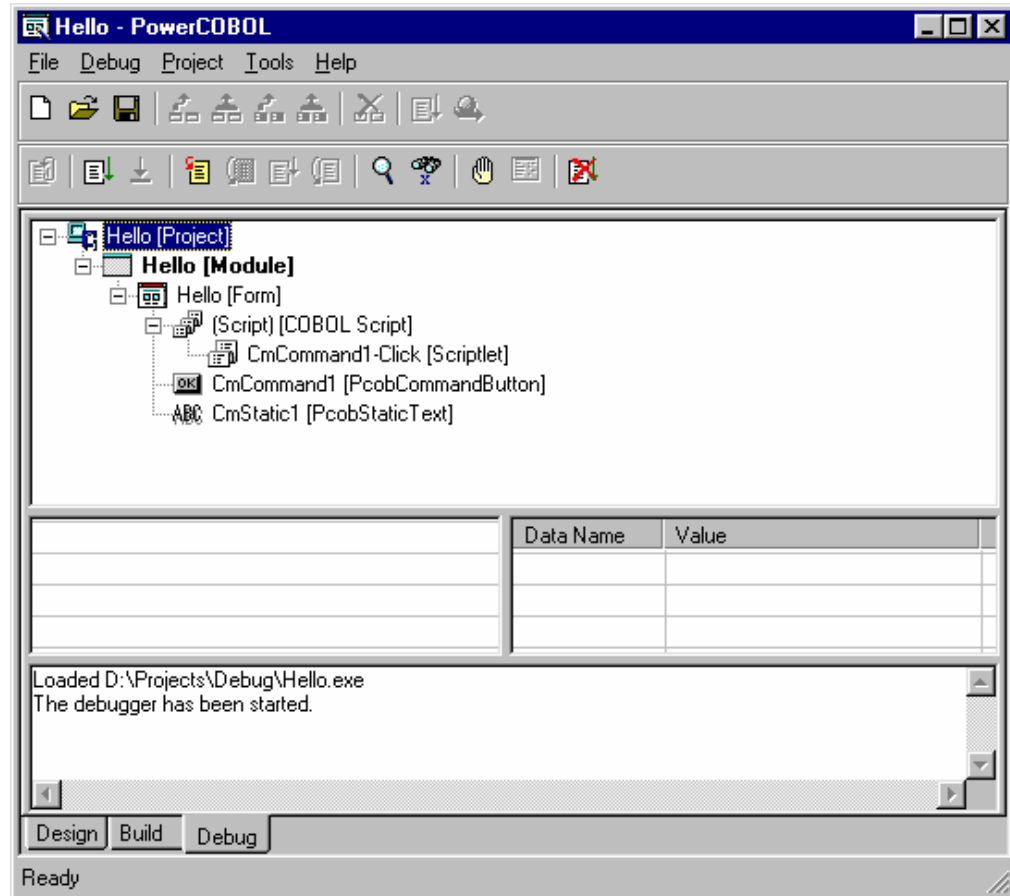


Figure 7.3. The Debug tab in the main PowerCOBOL Project Manager window

Main Debugger Window

The main Debugger window is displayed in figure 7.3. This window consists of a new tab that has been added to the main PowerCOBOL Project Manager window. Note that the two previous tabs - Design and Build - are still present. You may at any time during the debug process select one of these other tabs to view related information.

The Design tab is of special interest, as it still displays all of the application's objects in the Project Manager Window and each object's current properties when selected.

Selecting the Design tab does not terminate the Debugger, and you may step freely back and forth between these facilities.

Note that the application's project view is still visible in the upper left-hand windowpane. At any time during debugging, you can select procedures (scriptlets) from the project view to open them up in an Editor window.

While debugging, when you step into one of these procedures for execution, it will automatically be opened up in an Editor window to allow you to step through its associated event procedure code.

In the center right windowpane of the Debug tab, there is an area for listing Data Name and Value. The PowerCOBOL Debugger allows you to watch any number of data items in order to monitor their values as the application executes. This centralized area in the window displays all currently "watched" data items and their current values.

To the immediate left of the Data Name area is an empty windowpane with horizontal lines. During execution of the application, the program currently being executed and the specific procedure currently being executed will be displayed here, along with the current line number of each procedure awaiting execution.

The bottom windowpane is an information area. Debugger and run-time messages will be displayed here, including any error messages. As shown in figure 7.3, this area contains information indicating the name of the application executable that has been started in the Debugger.

In the very bottom left hand corner of the Debugger window, the word "Ready" is displayed. It is very important to keep an eye on this area of the Debugger window. "Ready" means that the Debugger is currently paused and ready for the user to issue a command. When the Debugger relinquishes control back to the application's user interface and is awaiting an application event to take place, the word "Running" will appear.

Whenever you see the word "Running" here, you will not be able to issue any execution instructions in the Debugger, as the application is currently executing and awaiting user input.


You may still, however, use some of the Debugger commands such as setting breakpoints. You may also select Break from the Debug menu to pause execution at the current statement.

Once the Debugger has been started and the application loaded, the application is immediately paused awaiting the first execution instruction. At this point, you may begin executing the application under the Debugger, or you may set breakpoints and/or select data items to watch (monitor) before starting execution.

Setting Unconditional Breakpoints

You may set unconditional breakpoints on any executable statement within an application. You may also set unconditional breakpoints on any event that may occur, such as a user clicking on a button.

You may set unconditional breakpoints from the main Debugger window, or from any Debugger Editor window. The Editor window will appear for a particular procedure when that procedure is invoked. In both cases, Set Breakpoint can be found in the Debug menu.

Additionally the Set Breakpoint icon -  may be selected directly from the main Debugger window.

When you invoke the Set Breakpoint option, you will be presented with the following dialog box:

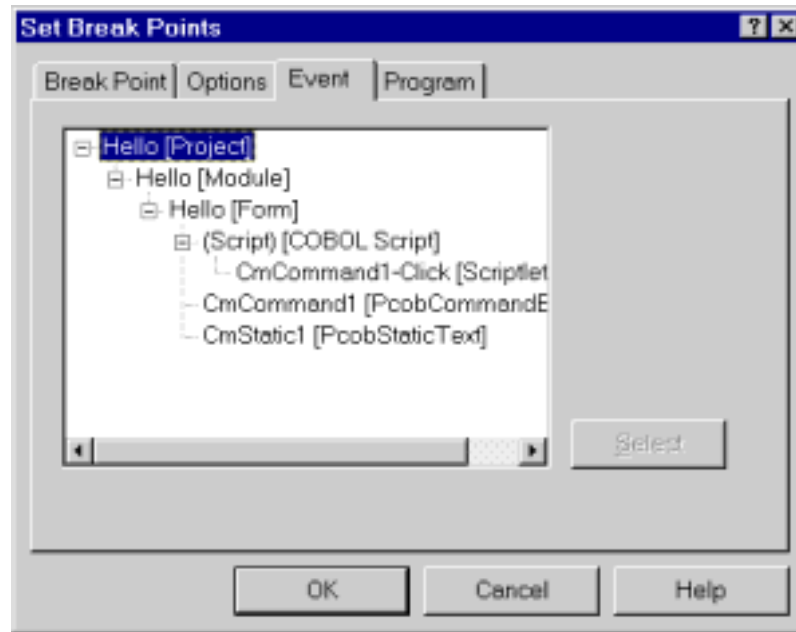


Figure 7.4. The Set Break Points dialog box

If you wish to set a breakpoint on a specific event occurrence, make sure the Event tab is currently selected. Then expand the project to show all event procedures (scriptlets). Select the specific scriptlet you wish to set a breakpoint for and click on the Select button. Now click on the Break Point tab in the dialog window.

The following dialog box is displayed:

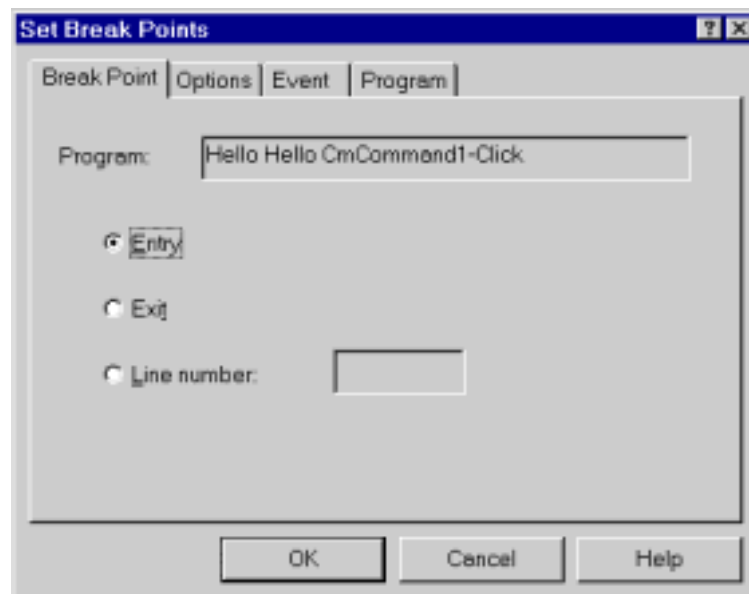


Figure 7.5. The Set Breakpoint dialog box

Here you may specify where in the event procedure code you want the breakpoint set:

Entry - specifies you want execution to stop on the first execution statement in the event procedure.

Exit - specifies that you want execution to stop on the last execution statement in the event procedure.

Line - allows you to specify a specific line number to stop execution on within the event procedure.

As opposed to setting an unconditional breakpoint on a specific event, you may optionally set a breakpoint on a specific program in your application. Remember that not all programs within your application must be event procedures (associated with specific GUI events). Instead you may have any number of sub-programs that perform tasks such as load data files, compute tables, etc.

If you wish to set an unconditional breakpoint on one of these non-event procedure programs, you do this by selecting the Program tab in the Set Breakpoint dialog box shown in figure 7.4. Clicking on this tab will present you with the following dialog box:

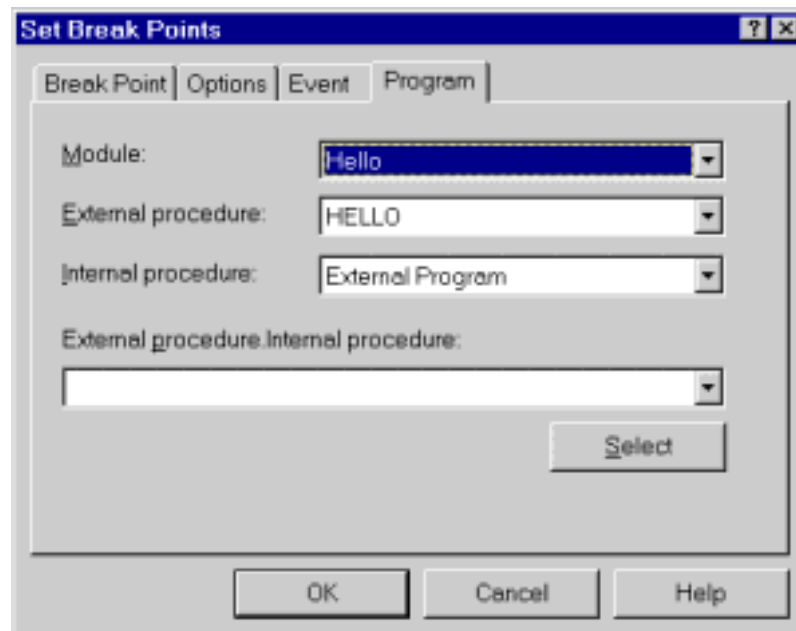


Figure 7.6. Setting a breakpoint on a program

You may select a program, and specify a specific procedure within the program to break on.

NOTE

You may also access the Set Breakpoint dialog box by right clicking on any executable line of the procedure code in the Editor window and selecting Set Breakpoint from the pop-up menu.

Setting Conditional Breakpoints

You may set conditional breakpoints in a manner similar to setting unconditional breakpoints.

You use the Option tab in the Set Breakpoint dialog box to specify the condition you want to break on. For example, if in the above noted application you want to break on the statement that causes the data item "Current-Count" to be greater than 5, you would enter the following:

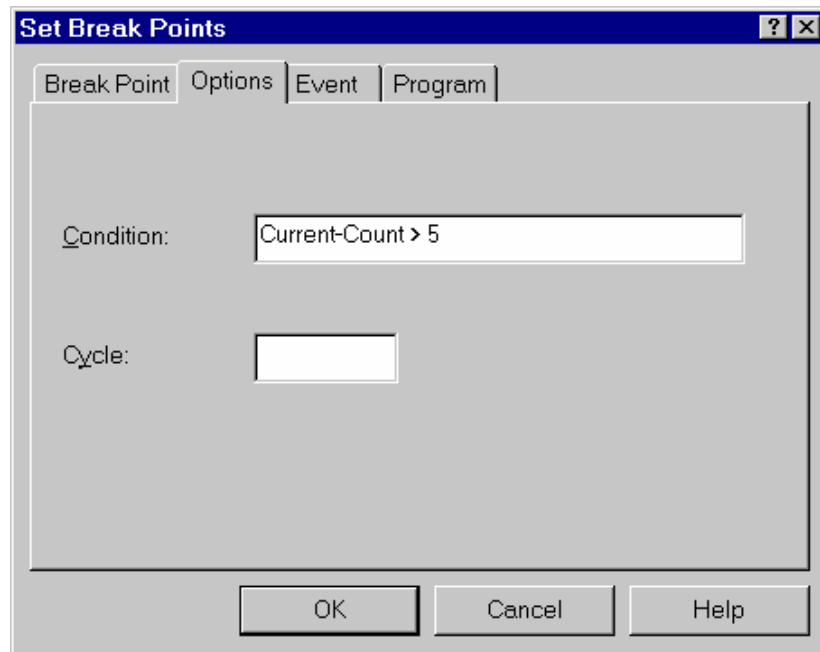


Figure 7.7. Setting a conditional breakpoint

You may set any number of conditional and/or unconditional breakpoints in the PowerCOBOL Debugger.

Cycle breakpoints specify that execution is to pause after a specified section of the application cycles a certain number of times. This means, for example, that if you want to allow the procedure to execute 5 times and break on the 6th iteration, you set the cycles option for 6.

Reviewing, Disabling, and Deleting Breakpoints

You may at any time review the current breakpoint status in an application by selecting Breakpoint List from the Debug menu. This will bring up a list of all currently specified breakpoints as follows:

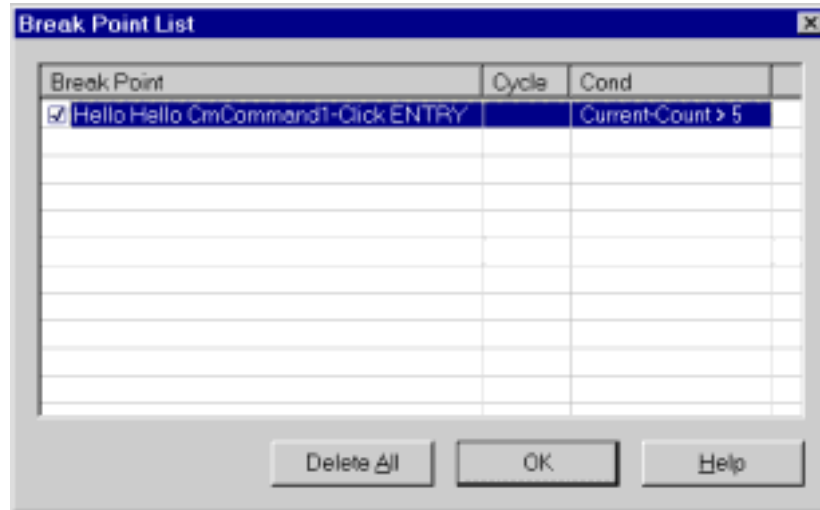



Figure 7.8. The Breakpoint List dialog window

NOTE: You may also use the Breakpoint List icon -  in the main Debugger window to bring up this list as well.

The Breakpoint List dialog box shows all breakpoints currently specified for the application. If there is a check mark in the check box to the immediate left of a breakpoint, it means that the breakpoint is currently active. If you wish to save a breakpoint specification, but need to deactivate it, simply uncheck the check box to the left of it.

To the right of a breakpoint you will see any related conditional breakpoints and/or cycle breakpoints currently specified.

If you wish to delete a breakpoint, right-click on it with the mouse and select Delete from the pop-up menu.

This same pop-up menu also allows you to access the breakpoint's properties, or to jump to the actual place in the application corresponding to the breakpoint. Note that "jump" does not mean execute. Instead, it means "show me the place in the code where this breakpoint is set".

Another related feature of the PowerCOBOL Debugger is the ability to force a break during application execution.

For example, if you select Go from the Debug menu, your application begins to execute quickly without showing you any of your source code statements.

If you have not set any breakpoints, your application will simply continue to execute until you close it. If you decide, however, that you need to break into the source code at a particular point in the execution phase, you can select Break from the Debug menu. This tells the Debugger to pause execution immediately.


If you currently have an Editor window open showing your procedure event source code, control will be returned back to this window and you may step through the source code statements.

If you do not have an Editor window open with the current event procedure source code, your application will be suspended. An Editor window is not opened automatically by selecting Break. If you want to step into the source code for the current event procedure, select Step Into from the Debug menu (or click on its corresponding icon).

Monitoring and Changing Data Items

The PowerCOBOL Debugger provides the capability to monitor (watch) any number of data item values during execution, and to dynamically modify any data item's value on the fly.

Data items being monitored are shown in the middle right pane of the Debugger window. You can add a data item to the list of items being monitored by selecting Watch from the Debug menu.

NOTE: You may access the Watch option using its icon  in the main Debugger window.

When you select the Watch option, the following dialog box is displayed:

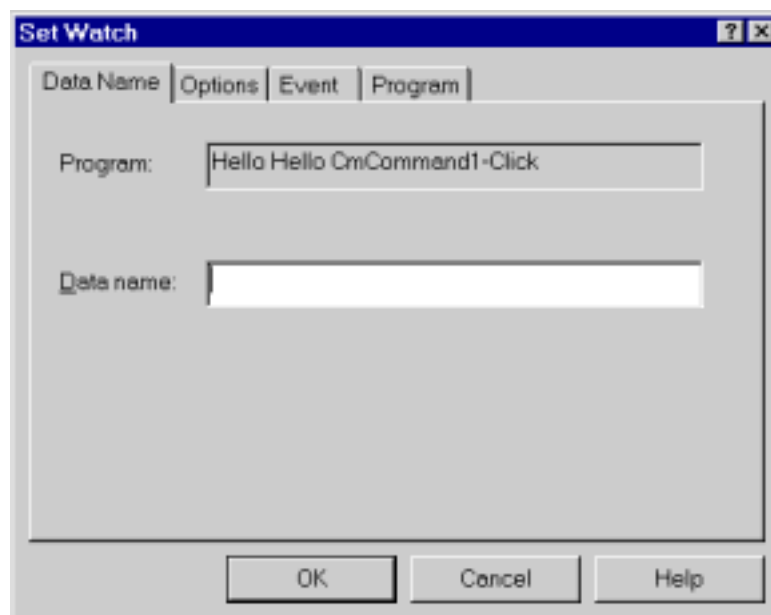



Figure 7.9. The Set Watch dialog box

Simply enter the name of the data item in the selected program you wish to monitor (watch) and click on the OK button. The data item name and its current value will appear in the main Debugger window.

The Option tab allows you to optionally specify that you wish to monitor the data item in hexadecimal.

You may monitor any number of data items using the Watch option.

If you wish to change a data item's current value, select Quick Watch from the Debug menu.

NOTE: You may access the Quick Watch option using its icon  in the main Debugger window, or by right clicking the mouse on any item in the current Watch list and selecting Quick Watch from the pop-up menu.

Once you select Quick Watch from the Debug menu, the following dialog box is displayed:

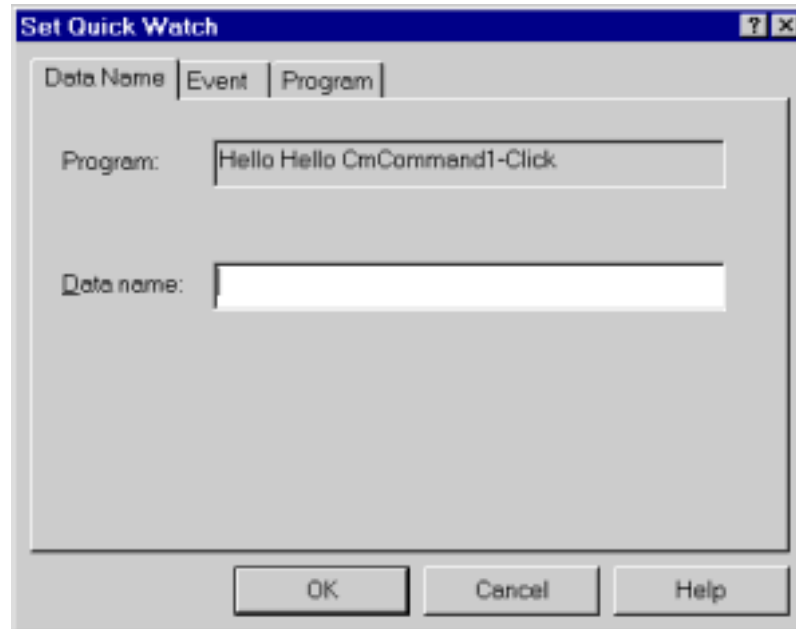


Figure 7.10. The Set Quick Watch dialog box

Simply enter the name of the data item you wish to access.

NOTE: If you are viewing the source code containing the data item you wish to watch or quick watch, you can simply highlight the data item name, right-click on it, and select either Watch or Quick Watch from the pop-up menu.

Once you have specified a valid data item name in the Set Quick Watch dialog box, or have right clicked on a currently watched data item and selected Quick Watch from the context menu, the following window will be displayed:

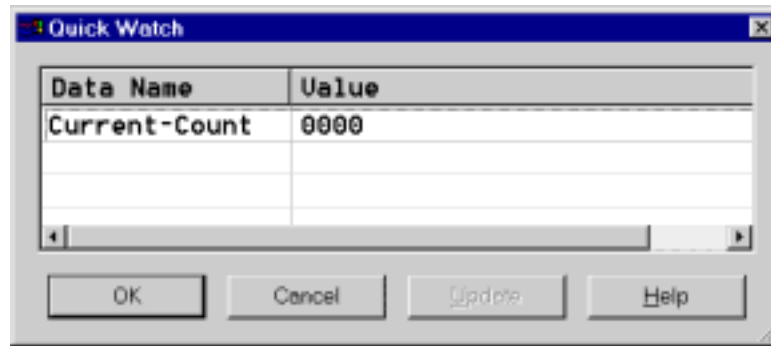


Figure 7.11. The Quick Watch window

To change the current value of the data item in the window, simply click on the value field and type in the new value. Then click on the OK button. The new value will take effect immediately.

To change the value of a data item currently contained in the watch area of the main Debugger window, simply right-click the mouse on the data item's name and select Quick Watch from the pop-up menu.

Controlling Application Execution

The PowerCOBOL Debugger provides multiple options for executing an application. You may execute a single statement at a time, an entire procedure in a step, execute multiple sections or procedures stopping on breakpoints, or simply execute the entire application without viewing any source code.

Making good use of breakpoints will help you balance the overall application debugging view that you desire.

When you start the Debugger, your application is loaded and prepared for execution. Actual application execution does not begin, however, until you instruct the Debugger to start.

This initial pause in the application's execution startup allows you to set any initial breakpoints or to select data items to monitor (watch) during execution.

Once you are ready to begin execution, you can either select Step Into or Go from the Debug menu.

The Step Into option initiates the application and will display the PowerCOBOL Run-time Environment Setup window. Once you click on the OK button in this window, you will be positioned on the first execution statement prior to the main form (window) being displayed. If you did not write any event procedures to execute before the main form is displayed, you will be presented with the main form. When you perform an action on the main form that generates an event for which you have created an event procedure, you will then be positioned at the first line of that event's execution code, and execution will be paused.

The Go option will begin executing your application without displaying any of its source code until it encounters a breakpoint, or until the GUI requires user interaction.

Unless you have created one or more event procedures to execute before your application's main form (window) is to be displayed, you should first see the main form on the screen.

If you have instead specified an event procedure to be invoked prior to the main form being displayed (for example, maybe you need to read a data file to populate fields on the main form before it is displayed), the Debugger will display your event procedure code.

Whenever the Debugger presents you with any of your application source code, it will do by displaying it in a live Editor window as shown in the following example:

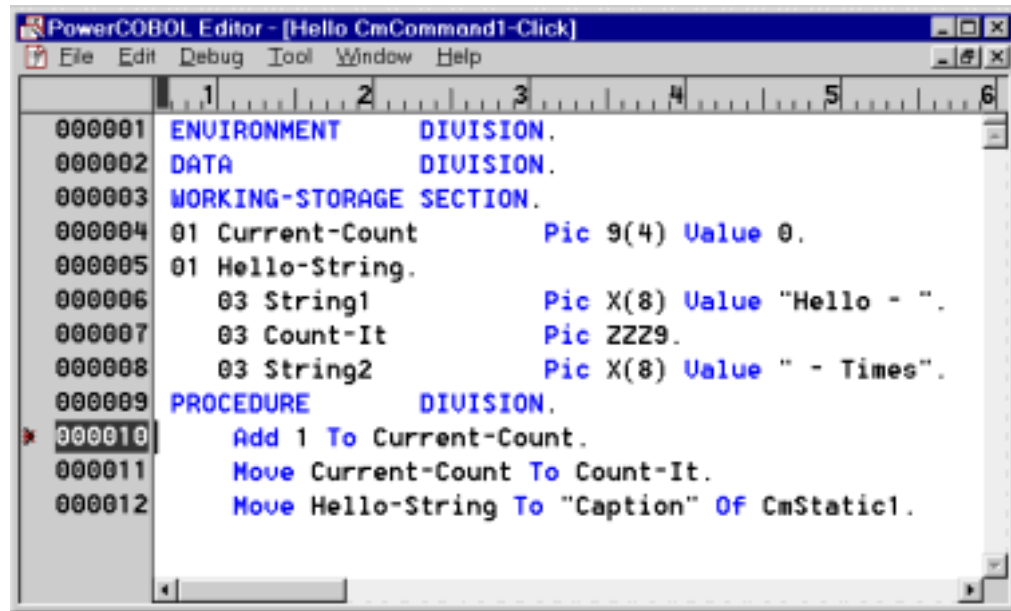


Figure 7.12. A live Editor window invoked by the Debugger

Within this window, you may control execution, set and delete breakpoints, select data items to be monitored (watched), and even make source code corrections. Source code corrections will not take effect until you quit the debug session and rebuild the application.

Whenever an Editor window is opened up for an event procedure, the line of procedure code about to be executed next will be designated with a ">" to the left of its sequence number.

As you step through execution statements, the ">" will move to the next execution statement to show you where you are currently positioned in the logic flow of your source code.

If you set a breakpoint on a line of code, it will be denoted by an "*" to the left of that line of code's sequence number.

One of the features of the PowerCOBOL Debugger is its support of opening multiple live Editor windows containing multiple event procedures. This allows you to examine source code and current data item values in sections of your application that may not currently be executing.

Once you have stepped into an event procedure and are looking at the source code in the editor window, you have four options in the Debug menu to control the next step in execution:

Step Into - this option will execute the next statement. If this statement causes a branch in the logic of your application (such as a CALL to another program, or a PERFORM on another paragraph), you will be branched in the logic to the next physical execution statement. Execution will be paused on this next execution statement. Use this option when you want to ensure that you step through the exact flow of logic.

Step Over - this option will execute the next statement. If this statement causes a branch in the logic of your application (such as a CALL to another program, or a PERFORM on another paragraph), you will not be branched. Instead, all of the statements in the called program or performed paragraph will be executed as a single step. You will then be positioned in the current event procedure on the next physical execution statement following the CALL or PERFORM that was just executed.

Run To Cursor – this option will execute up to the line you position the cursor on.

Run to Exit - this option will perform all of the statements in the current event procedure instantly (including any called programs or performed paragraphs) up to the last statement in the current event procedure. Execution will then be paused on the final execution statement without executing it. This allows you to examine the state of the current event procedure before it is exited.

Go - this option causes execution to start immediately and proceed until a breakpoint is encountered or you select Break from the Debug menu. The application will execute as if it is running normally at machine speed.

You may use these different options under various conditions to enhance your debugging productivity.

Examining Data Item values in the Editor under the Debugger

After you have begun execution under the debugger and you are currently stopped in execution in an editor window, you can use a very handy feature to examine the contents of any initialized data item.

To do this, simply hover the mouse pointer over the data item you wish to examine and its current value will pop up in a small window such as in the figure below:

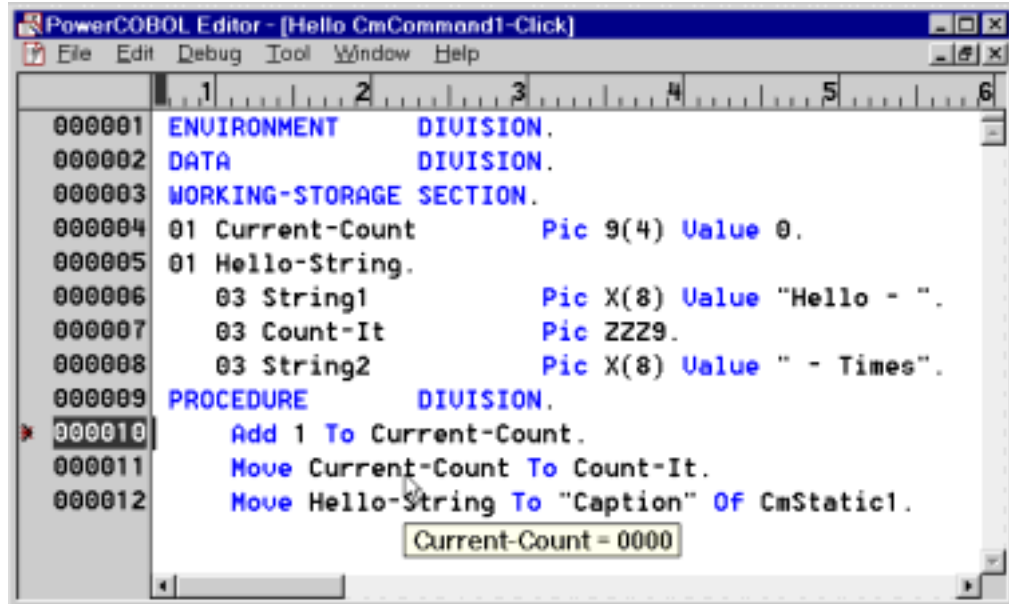


Figure 7.13. Hovering over a data item to display its current value

Diagnostic Reporting

You can use the diagnostic function when the application abends or outputs an error messages. The diagnostic information reports the module name, form name, event procedure name, line number and other useful information. PowerCOBOL utilizes the COBOL diagnostic function; so refer to the "COBOL Debugging Guide" for details.

The diagnostic function is automatically switched on whenever you build a module in debug mode. You may optionally select the diagnostic function in a release mode module by clicking the appropriate checkbox in the project's Properties Build tab as shown below:

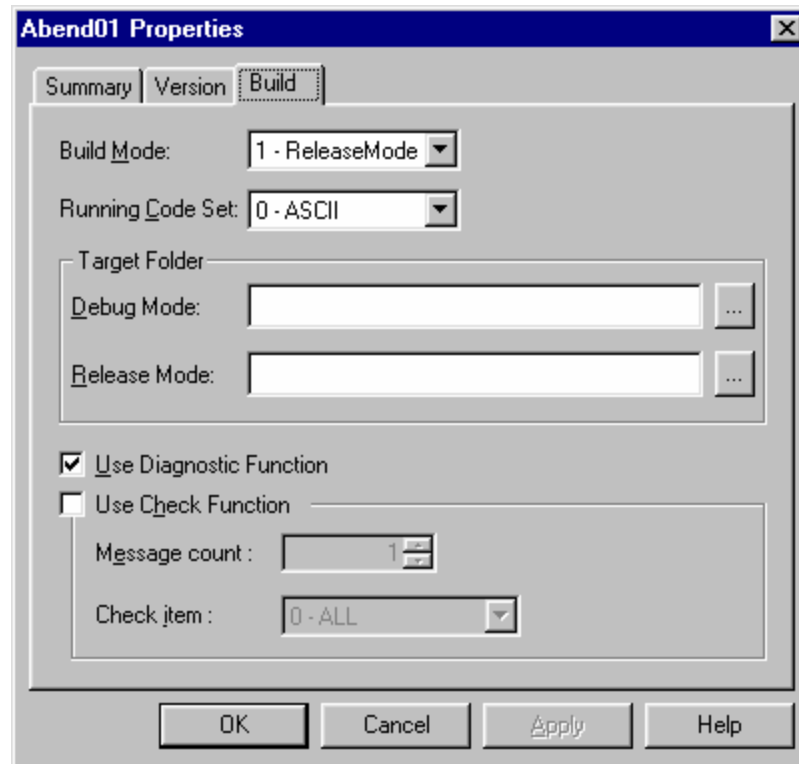


Figure 7.14. Turning on the Diagnostic Function

Setting up the Environment

Environment for Release Mode

Check "Use Diagnostic Function" in the "Build" tab of the project Properties dialog.

The size of the debug information file (FormName.SVD) is smaller if the "OPTIMIZE" compile option is specified.

Note that, in Release mode, you must move or copy the debug information files (*.SVD and *.PLE) to the folder in which the executable file exists if you move or copy the executable file to another folder.

Environment for Debug Mode

When you select Debug mode, "Use Diagnostic Function" is checked automatically.

If you are using programs from multiple projects, set the folder name, in which the debug information files (*.SVD and *.PLE) of the other modules reside, in the "Debug information file folder" field in the "Debug" tab of the module Properties dialog.

Note that, you cannot use the diagnostic function effectively if you move or copy the executable files to another folder, or you do not execute within the PowerCOBOL development environment, because PowerCOBOL cannot find the debug information files.

Additional Information

- The .pli file that is output in the folder contains information about the event procedure lines (PowerCOBOL Line Information). If you remove this file, PowerCOBOL outputs only the COBOL level information - not the event procedure line information.
- PowerCOBOL outputs the following files to support the provision of diagnostic information:

Information file	File type	Output folder (Release mode)	Output folder (Debug mode)
COBOL Debugging information file	SVD	Target folder	Working folder
PowerCOBOL line information file	PLI	Target folder	Target folder

- If you specify the compile option "OPTIMIZE", the size of the .svd file becomes smaller, and you cannot debug the project in debug mode.

Understanding the Report

PowerCOBOL outputs information (lines shaded with text in bold) to this part of the report:

```

...

<<Detail>>
Thread ID      : FFE8BC61
Register       : EAX=00000000  EBX=0085F814  ECX=00000000  EDX=00000000  ESI=0085F7D8
                : EDI=0085F8FC  EIP=00401377  ESP=0085F778  EBP=0085F8C4  EFL=00010283
                : CS=0167  SS=016F  DS=016F  ES=016F  FS=12AF  GS=0000
Stack Commit   : 00005000 (Top:00860000, Base:0085B000)
Instruction    : Address  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +a +b +c +d +e +f
                00401367 00 0F BF 05 42 60 40 00 99 0F BF 0D 3A 60 40 00
                FAULT ->00401377 F7 F9 66 89 45 DA 0F BF 45 DA B1 04 B5 00 8D 3D

Module File    : C:\SAMPLES\RELEASE\MAIN.EXE
Section Relative Position : .text+00000377
Export Relative Position : MAINFORM+00000343
Symbol Relative Position : MAINFORM+00000377
Compilation Information : ASCII, SINGLE THREAD, NOOPTIMIZE
PowerCOBOL Project File : C:\Samples\Abend01.ppj
Module : Main
Form : MainForm
Scriptlet : MainForm-Click
Line : 8

<Call Stack>
[ 1]-----
Module File    : C:\SAMPLES\RELEASE\MAIN.EXE
Section Relative Position : .text+000001F8

```

```
Export Relative Position : MAINFORM+000001C4
Symbol Relative Position : MAINFORM+000001F8
Compilation Information : ASCII, SINGLE THREAD, NOOPTIMIZE
External Program/Class : MAINFORM
Source File : MainForm.PRC
Source Line : 35
```

```
...
```

When "Use Diagnostic Function" is checked in the Build tab of the project Properties dialog, PowerCOBOL adds the following information to the report. Refer to the "COBOL Debugging Guide" for an explanation of the standard information contained in the file.

PowerCOBOL Project File

Provides the name of the PowerCOBOL project file containing the module that raised the application error. If the PowerCOBOL line information file cannot be found, "Unknown" is inserted.

Module

Provides the name of the module that raised the application error.

Form

Provides the name of the form that raised the application error.

Scriptlet

Provides the scriptlet name that raised the application error.

Line

Provides the line number that raised the application error. This is the line number within an event procedure or the relative line in an included COBOL file.

Table 7.1 Information provided at a break point and in the diagnostic report

Information	Break point			Diagnostic report			Notes
	Event procedure	External COBOL file	Form's common program	Without Power-COBOL line information file	Without COBOL debugging information file	Without both information files	
Module File	OK	OK	OK	OK	OK	OK	
Section Position	OK	OK	OK	OK	OK	OK	
EXPORT Position	OK	OK	OK	OK	OK	*1	
Symbol Position	OK	OK	OK	OK	OK	*1	
Compilation Information	OK	OK	OK	OK	OK	*1	
PowerCOBOL Project File	OK	OK	OK	*2	-	-	Specific to PowerCOBOL
Module	OK	OK	OK	-	-	-	Specific to PowerCOBOL
Form	OK	-	OK	-	-	-	Specific to PowerCOBOL
Scriptlet	OK	-	-	-	-	-	Specific to PowerCOBOL
Line	OK	OK	OK	-	-	-	Specific to PowerCOBOL
External Program / Class	-	OK	-	OK	-	-	
Internal Program / Method	-	OK	-	OK	-	-	
Source File	-	-	-	OK	-	-	
Source Line (COBOL)	-	-	-	OK	-	-	

*1) For this information you need to specify the link options, /DEBUG, /DEBUGTYPE:{COFF | BOTH}

*2) This information becomes "Unknown".

Check Function

The Check function was introduced in version 6.1 of PowerCOBOL. It corresponds to the NetCOBOL compiler's "Check" function. It is typically used in debugging mode as it causes additional code to be generated by the compiler and performance is degraded as a result.

You should always rebuild your application when changing the Check option.

The following sub options are available under the Check function when enabled:

Message Count

Sets the number of times certain runtime error messages are to be ignored, thus allowing execution to continue.

Check Item

Selects the target to check.

Chapter 8. Developing Your First ODBC Application

In this chapter, you will develop your first ODBC (Open Database Connectivity) application using PowerCOBOL.

NOTE: This exercise requires the NetCOBOL Professional Edition or Enterprise Edition.

In the past, most PC database vendors provided highly proprietary application programming interfaces (API's) as a mechanism for accessing their database management systems (DBMS). These API's tended to vary widely, and were generally designed for the C and C++ programming languages.

Legacy COBOL compiler vendors were slow to provide support for even the most common DBMS on the PC. This support generally came in the form of a limited function COBOL API that was mapped onto the DBMS' API, and in some cases, SQL preprocessors.

As both SQL and database systems have evolved, the need for a more standardized and generic method of accessing them has grown.

ODBC is a defined interface that attempts to provide a highly generic API into any database system that provides ODBC compliant drivers. Just about every database system available on the PC today provides ODBC drivers for a variety of platforms.

ODBC also provides a transparent interface for accessing database servers physically running on other machines on a network. This makes it ideal for developing client/server applications that require database functionality.

In this chapter you will develop an application that will use ODBC technology to access a simple Microsoft Access database system.

Overview

The application you are going to develop will use a single window (form) to display three data fields. At run-time, the application will connect to a small Microsoft Access database (Test.mdb), and will interact with the "Employee" table. It will deal with three fields in the "Employee" table - "EmployeeNumber" (the primary key), "FirstName", and "LastName". You will create a read-only application that connects to the database and allows you to browse data. Then you will enhance the application to add, delete, and update database records.

This application will demonstrate the value of using PowerCOBOL's high-level ODBC programming functions (*methods*) to simplify development.

Software Prerequisites

In order to proceed with this exercise, you must have already installed 32 bit ODBC support on your Windows 95, Windows 98, Windows Me, Windows NT or Windows 2000 system, along with the Microsoft Access 7.0 (or higher) Driver.

You will additionally be required to have Microsoft Access Version 7.0 (or the Access run-time engine files) installed.

In order to verify whether your system has the proper support installed, look in your Windows Control Panel for the [ODBC] Data Source icon (in the Administrative Tools sub-group in Windows 2000). Double-click the left mouse button on this icon. Select the ODBC Drivers tab and verify that the Microsoft Access Driver (*.mdb) version 3.5 or higher is installed.

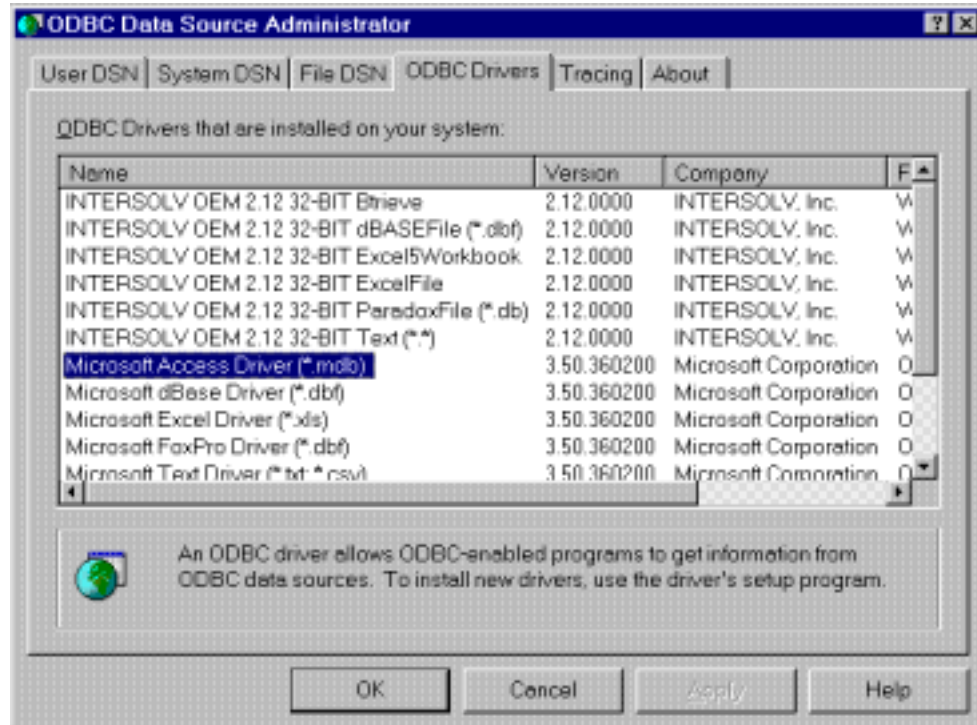


Figure 8.1. Checking for MS Access ODBC driver under Windows NT 4.0

When installing Microsoft Access (either from the Access installation CD or the Office Professional CD), the ODBC drivers are not selected by default - you must manually select these options.

If you do not have these installed, they may both be found on the Microsoft Access or Microsoft Office Professional installation CD. Note that when installing Microsoft Access, you must look under the installation options for Data Access and select Microsoft Desktop ODBC Drivers, which are not selected by default.

You will also require a sample Access database called "Test.mdb". This file can be found in the following directory:

C:\Program Files\Fujitsu NetCOBOL for Windows\Samples\PowerCOBOL\DBAccess

Once you have verified installation of ODBC, Microsoft Access, the associated drivers, and the test database, you are ready to move on with the exercise.

Defining an ODBC Source for Your Application

The first development task that needs to be accomplished prior to starting PowerCOBOL is to define an ODBC Source for the application.

An ODBC Source can be thought of as simply specifying to the ODBC manager a symbolic name that you wish to use in your application to describe an ODBC connection using a specific ODBC database driver to access a physical database.

The following figures may or may not match what you see on your particular operating system (depending on what software you have installed), but you should be able to perform the following tasks by using the instructions provided. The following figures have been taken from the Microsoft Windows NT 4.0 environment. If you are running under Windows 95, Windows 98, Windows Me, Windows 2000, what you see may vary somewhat.

Bring up the Windows Control Panel and double-click the on the [ODBC] Data Sources item (this is in the Administrative Tools sub group of the Control Panel in Windows 2000). The main ODBC Data Source Administrator window is displayed:

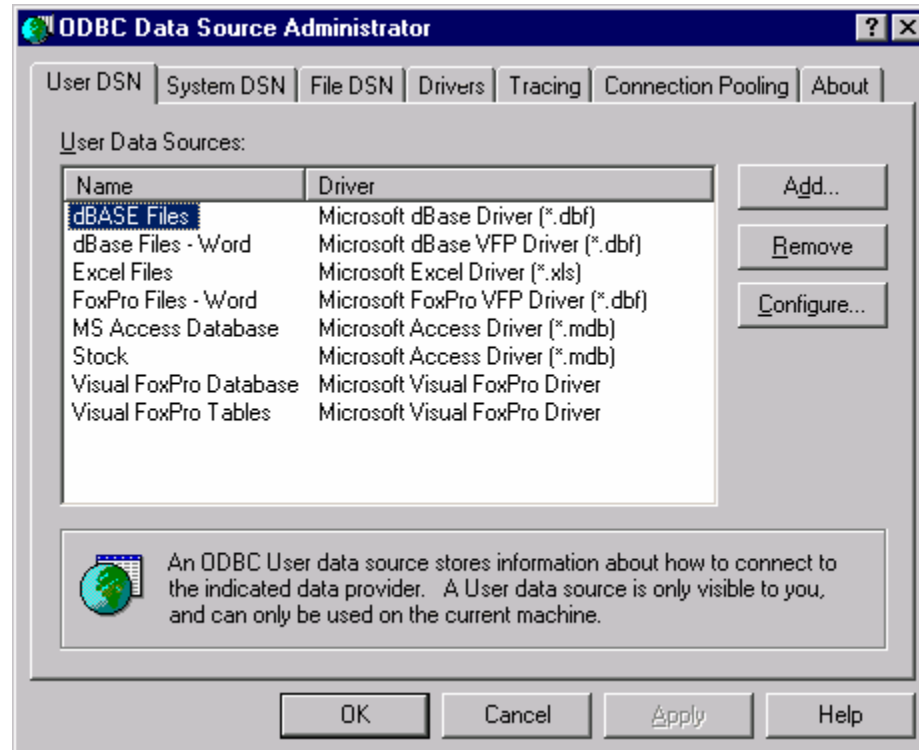


Figure 8.2. The main ODBC Source Administrator window

You will add a new user data source name (DSN). Make sure the User DSN tab is open, and click on the Add button. The Create New Data Source dialog box is displayed. Highlight the Microsoft Access Driver (*.mdb) as follows:

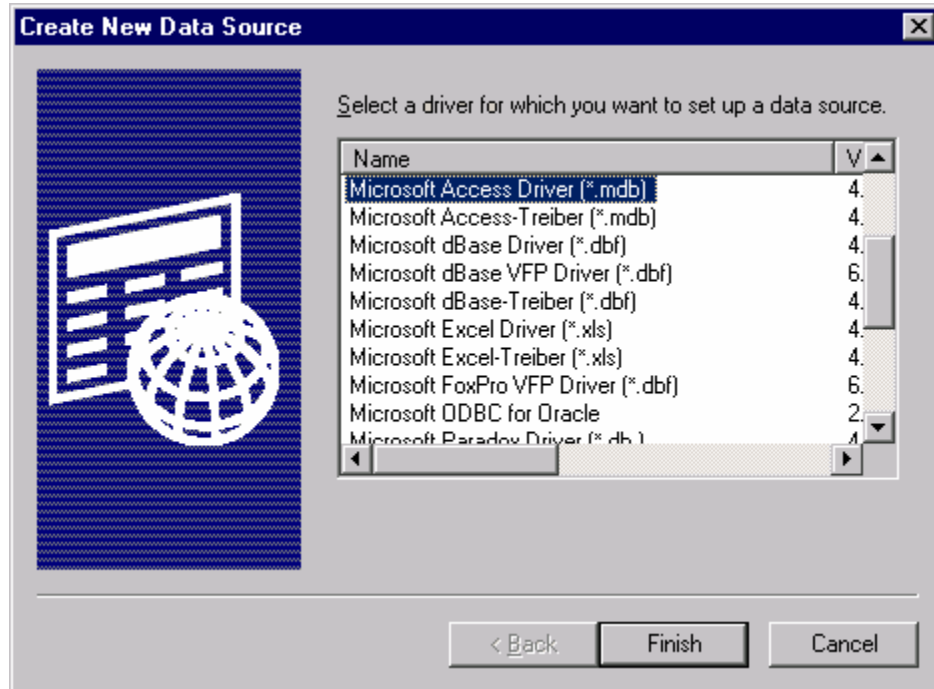


Figure 8.3. The Create New Data Source dialog box

Click on the Finish button.

The ODBC Microsoft Access Setup dialog box is displayed. Enter **MYODBC** for the data source name and **Fujitsu COBOL Test** for the description:

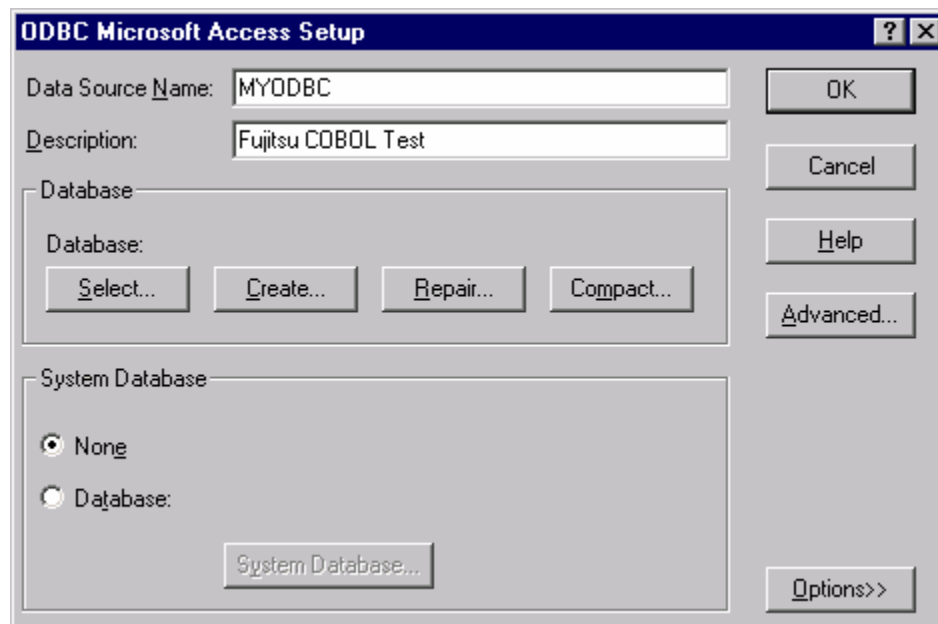


Figure 8.4. The ODBC Microsoft Access Setup dialog box

You also need to select a file (database) to bind this driver to. Click on the Select button in the ODBC Microsoft Access 97 Setup dialog box. The Select Database dialog box is displayed. Navigate through the directories and find the "Test.mdb" database file:

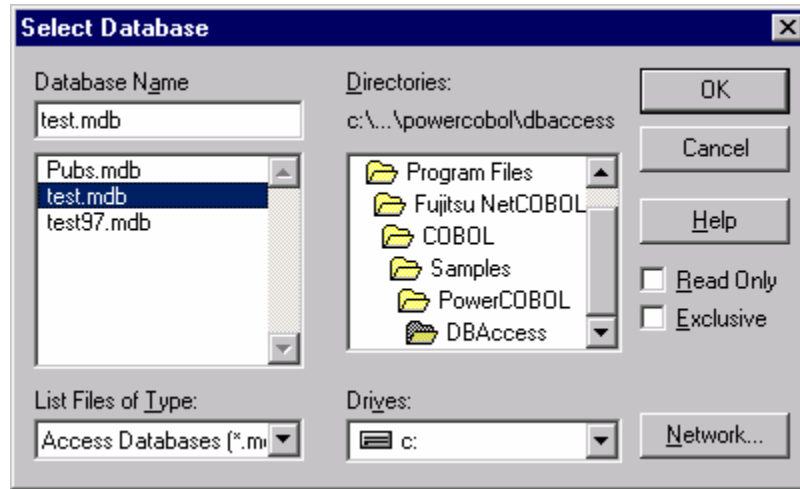


Figure 8.5. The Select Database dialog box

Select "test.mdb" and click on the OK button. This will take you back to the ODBC Microsoft Access Setup dialog box shown in Figure 8.4. Click on the OK button.

This will take you back to the main ODBC Source Administrator window. Note that "MYODBC" has been added to the list of User DSN's. Although you should not set any additional options for this exercise, you may wish to highlight "MYODBC" and click on the Configure button.

This will give you an idea of some of the additional options that are available when developing these types of applications. Once you are back to the main ODBC Source Administrator window, click on the OK button. This will exit the ODBC configuration facility. You have now defined the new ODBC Source (MYODBC) you are going to use from PowerCOBOL.

This is a generic database source using the standard MS Access ODBC driver, and is thus not specific to PowerCOBOL.

Beginning the Development Process

Bring up the PowerCOBOL Project Manager:



Figure 8.6. The PowerCOBOL Project Manager

Creating a New Project

You are now going to create a new PowerCOBOL project.

Select New Project from the File menu. The New Project wizard dialog box is displayed:

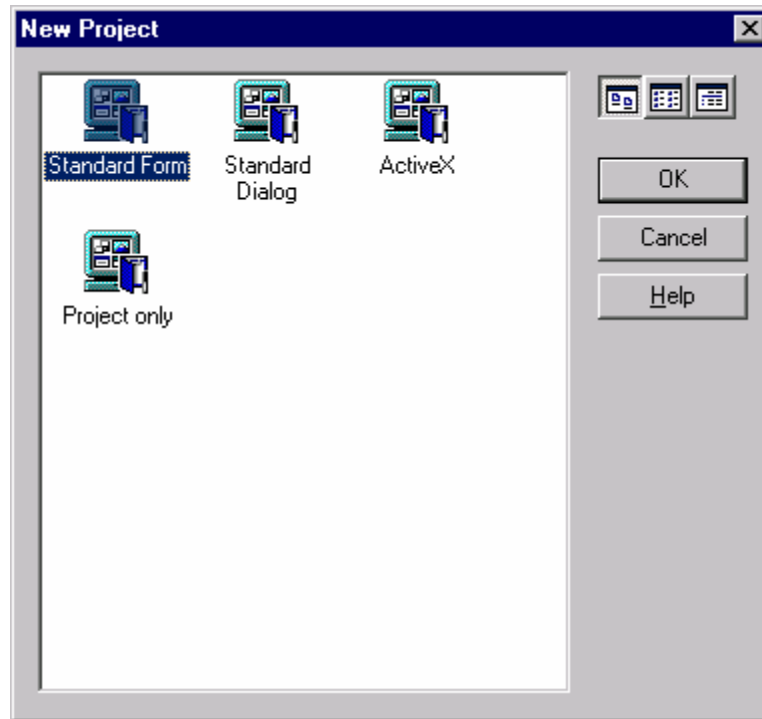


Figure 8.7. The New Project wizard dialog box

Make sure the first Standard Form icon is highlighted (this creates a project with a window that contains no controls), and click on the OK button.

You will now be taken back to the Project Manager main window, and a new project named "Untitled" will appear in the left windowpane.

Move the mouse to the project name in the left windowpane, right-click on it, and select Expand All from the pop-up menu. The Project Manager window should now appear as follows:

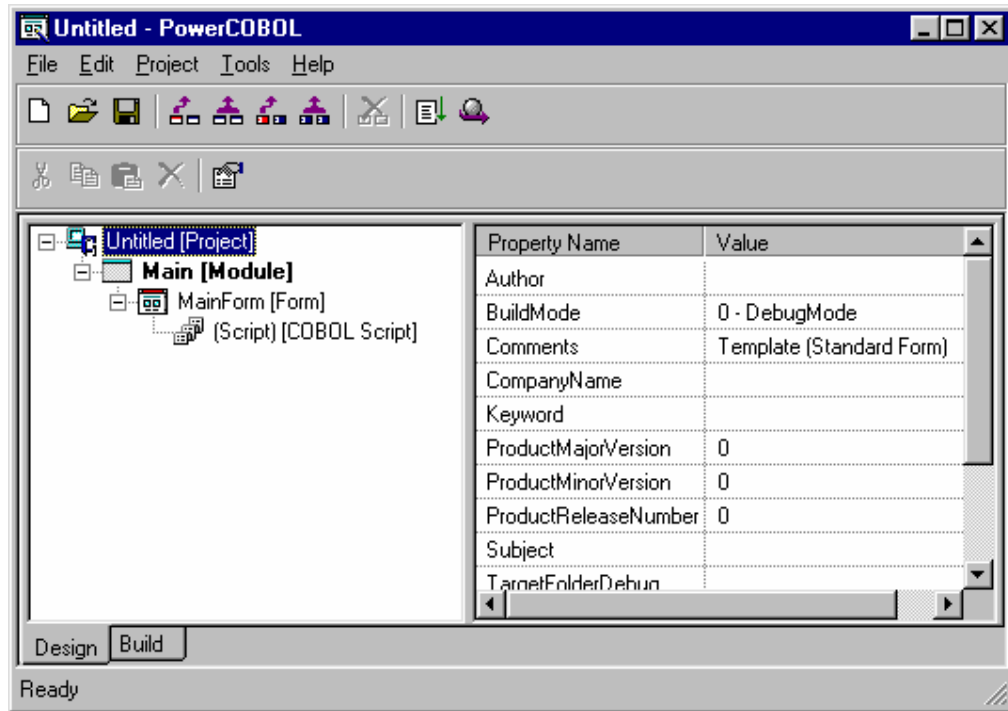


Figure 8.8. The Project Manager window with the newly created project displayed

PowerCOBOL has created a new project containing a single main application module named "Main", and a single form (window) named "MainForm". For the purpose of this exercise, you will retain and use these names.

Select Save As from the File menu. Change the name of the project from "Untitled" to "MYODBC" and navigate to the directory (folder) you wish to save the project in. You might even want to create a new directory (folder) to save this in. Click on the Save button to save the project.

The name of the project will now change from "Untitled" to "MYODBC" in the Project Manager window.

Developing the Graphical User Interface

You are now ready to begin developing the graphical user interface. You should already be familiar with the PowerCOBOL 'drag and drop' development paradigm.

It is important that you follow the process defined below, but given PowerCOBOL's flexibility, you are welcome to move and resize fields on the fly and add any small enhancements you may want to experiment with.

Begin by opening the MainForm form (window) in the Form Editor. You do this by right clicking the mouse on MainForm in the left windowpane of the Project Manager window and selecting Open from the pop-up menu. An alternative way of opening the Form Editor on MainForm is to select the Object sub menu from the Edit pull down menu in the Project Manager and clicking on the Open option. Also note that if you check the Object Default Action option which is found in the design tab of the Project Manager's Options dialog window (found under the Tools pull down menu), you can simply double-click on the name of a form to open it up in the Form Editor.

When MainForm is opened for the first time in the Form Editor, it should appear as follows:

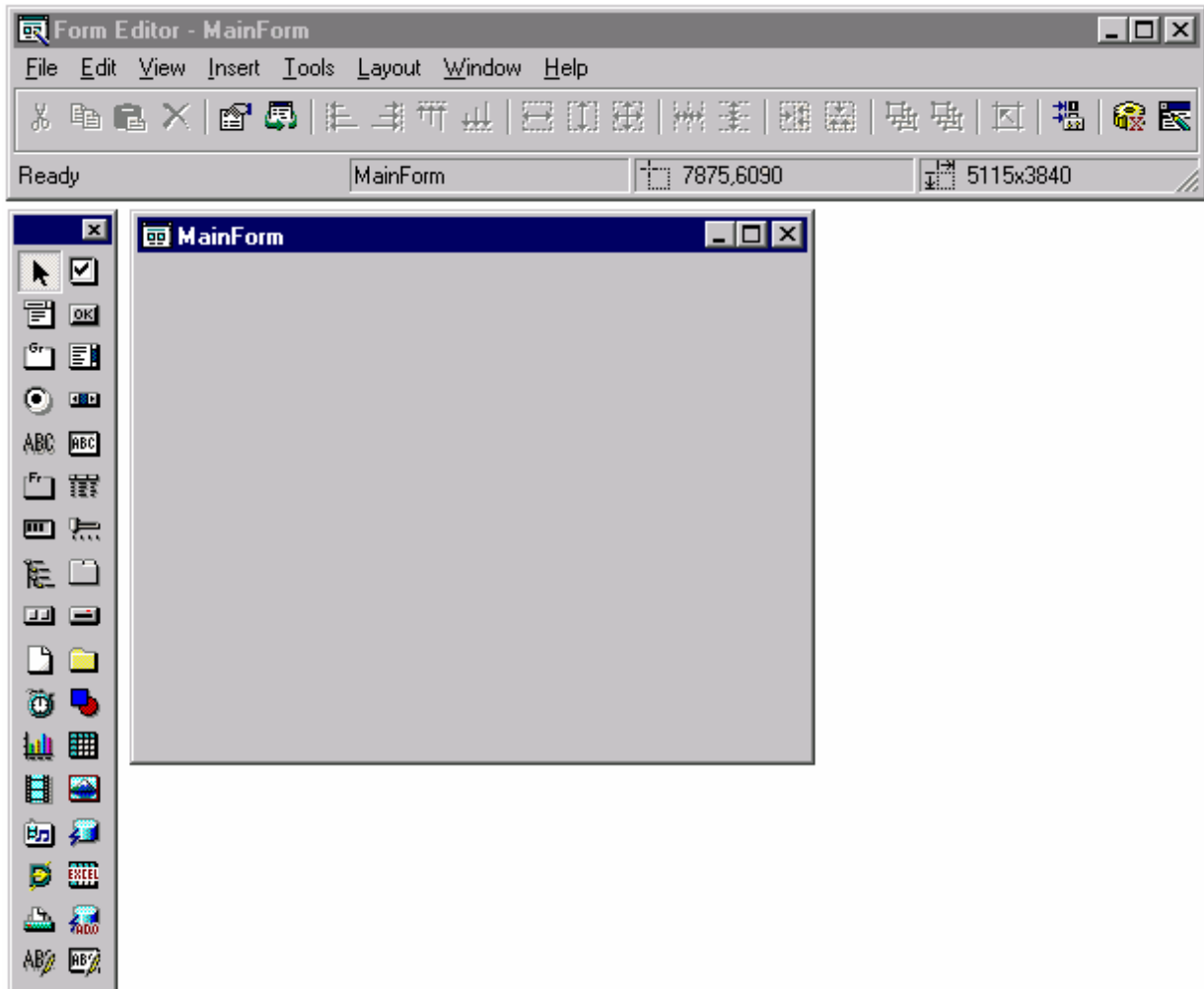


Figure 8.9. The Form Editor

MainForm is the main and only window that this application will use. You will thus place controls on MainForm using the drag and drop programming paradigm, and insert event procedures as needed.

The first control you will add to the form will be the ODBC support.


Find the DB Access Control icon  in the Toolbox palette on the left and click on it with the left mouse button. Move the mouse to the upper right corner of the form and click the left mouse button again to paste it on the form as shown in the following example:



Figure 8.10. The form after adding ODBC Support

It does not matter where you actually paste the DB Access Control on the form. It will not be shown as part of the form when the application is executed. Instead, the icon shown on the form is to illustrate that ODBC support is being built into this application.

Now you need to customize the ODBC support for this specific application.

Right-click the mouse on the DB Access Control icon on the form and select Properties from the pop-up menu.

This will bring up the DB Access Control Properties dialog box as follows:

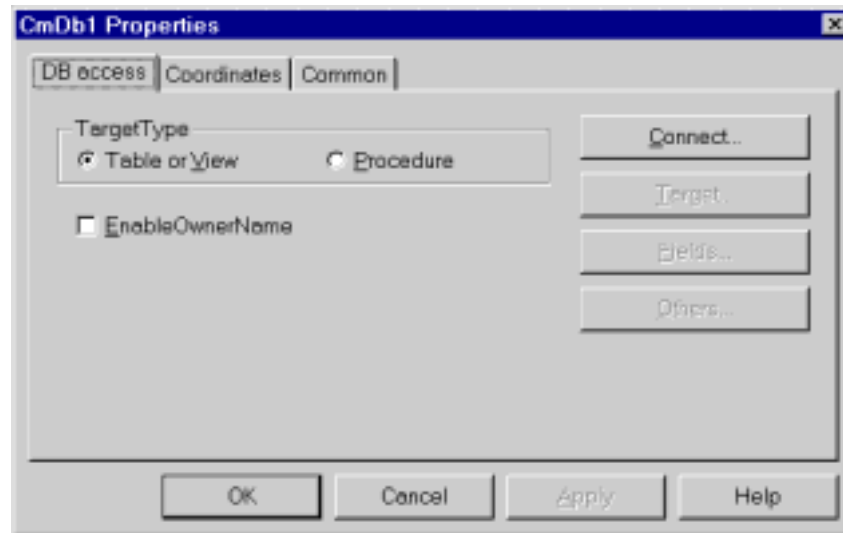


Figure 8.11. The DB Access Control Properties dialog box

You will note that PowerCOBOL has assigned a name of CmDb1 to this control by default.

Also note the TargetType property that defaults to Table or View. You may alternatively access SQL stored procedures in database systems that support such via this property. For now, however, leave the TargetType property set to Table or View.

You will now connect to the external data source name you previously defined under ODBC Data Sources at the beginning of this chapter. Click on the Connect button in the dialog box.

The Select Data Source dialog box is displayed. Select the Machine Data Source tab and highlight the MYODBC data source name as follows:

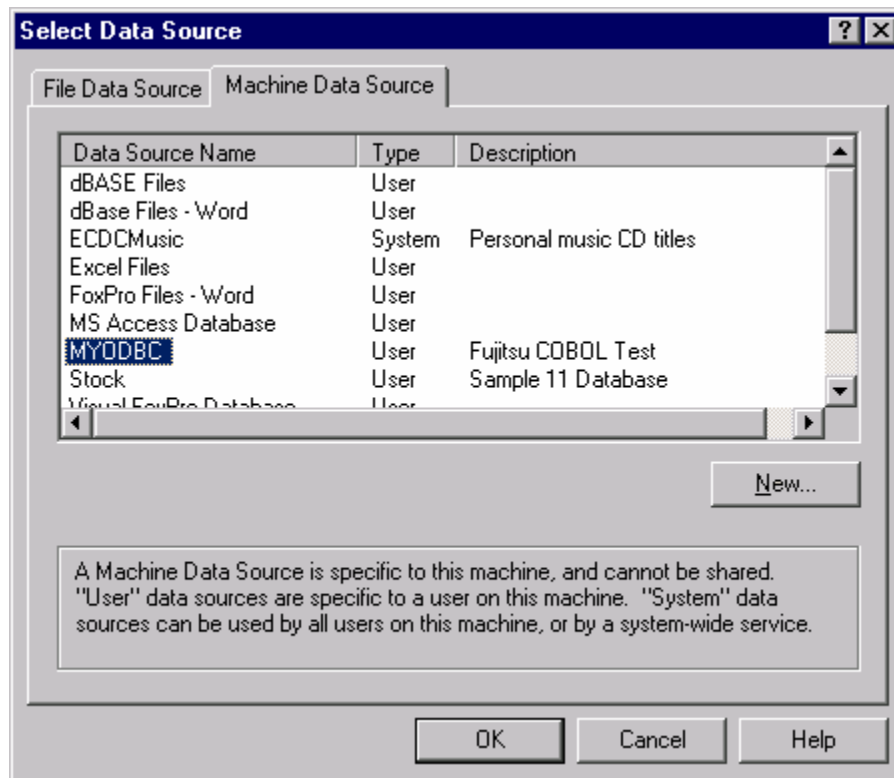


Figure 8.12. The Select Data Source dialog box

Note that you may actually define a new ODBC data source from here should you desire. In this case, you should have created an ODBC data source prior to accessing dialog window.

Click on the OK button. You have now connected your application with the specific ODBC Source driver that will access the "Test.mdb" database.

The DB Access Control Properties dialog box is redisplayed. Note that the Target button has now been enabled.

The "Test.mdb" database contains a single table named "Employee". Relational databases may contain multiple tables and/or custom views.

Click on the Target button to bring up the Select Table/View dialog box to specify the table(s) and/or views you wish to access from this application:

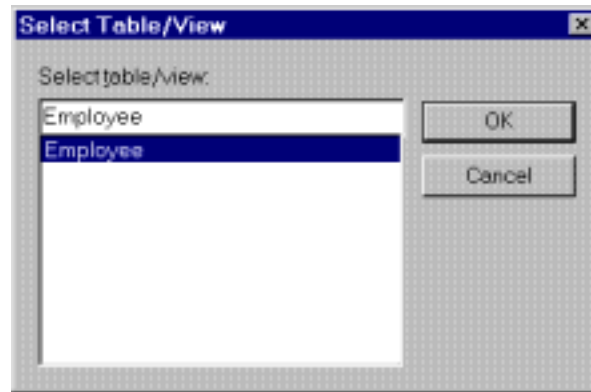


Figure 8.13. The Select Table/View dialog box

In this case, there is only one table available (Employee). Select it and click on the OK button to return to the DB Access Control Properties dialog box.

Note that the Fields button has now been enabled. Click on the Fields button. The Select Field dialog box is displayed. It shows you all of the individual database fields contained in the "Employee" table.

In some applications, you may not need to access all of the database fields. In this application, however, you will access all three fields, so you need to specify this.

If you want to add all of the database fields to the Selected fields list, as in this example, you can simply click on the >> button without selecting any individual field name first. You may alternatively select each field in the Field list box individually and then click on the > button to add it to the Selected fields list box one at a time.

When you have completed adding all three fields to the Selected fields list, the dialog box should appear as follows:

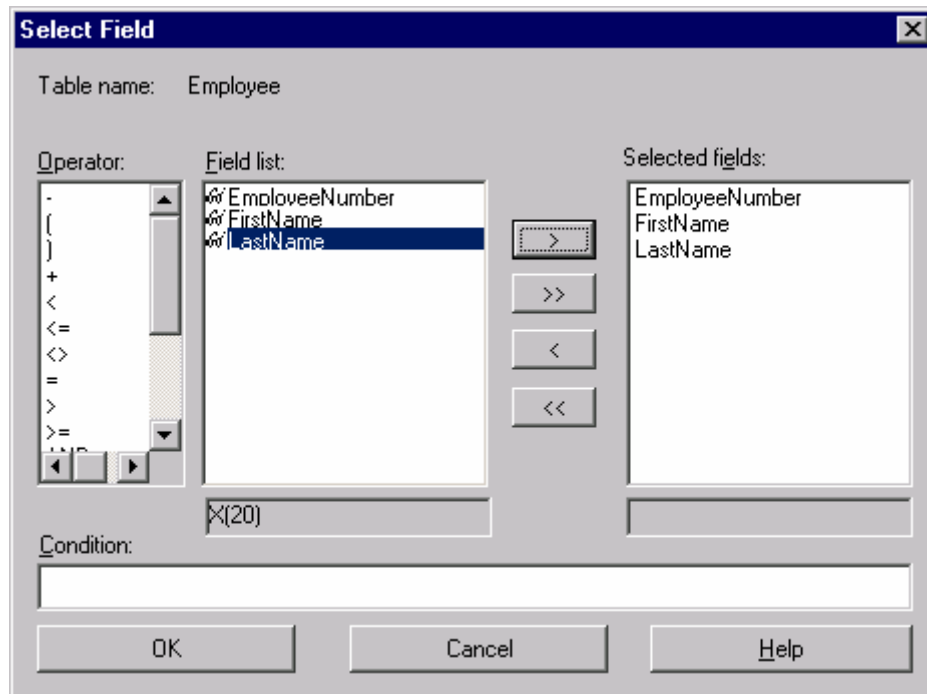


Figure 8.14. The Select Field dialog box with all fields selected

Do not select any of the operator options at this time. Click on the OK button to return to the DB Access Control Properties dialog box. The Others button has been enabled. Click on the Others button.

This brings up a dialog box containing other options for this ODBC connection.

Un-check the Read only check box. Now move the mouse to the Selected fields list box and double-click on "EmployeeNumber".

A small key icon will appear to the left of this field name, indicating that this will be the primary key field. Click on the OK button to close this dialog box and then click OK one more time to close the DB Access Control Properties dialog box.

You have completed the ODBC database definition and configuration phase of the project.

Completing Development of the Graphical User Interface


You will now finish developing the graphical user interface portion of the project. You will do this by placing three separate textbox controls (with separate static text controls as labels) on the form, along with two command buttons.

When you are finished with the steps described below, the form should appear as follows:



Figure 8.15. The completed graphical user interface

Creating the Data input/output fields

Find the TextBox Control icon  in the Toolbox palette and click on it with the left mouse button. You will use this textbox control as a data input and output field on the form. Move the mouse back to the sheet and drop the first edit field near the top center of the window.

Right-click the mouse on the new textbox control and select Properties from the pop-up menu that appears. Alternatively you can select Properties from the Edit menu. The TextBox Properties dialog box appears:

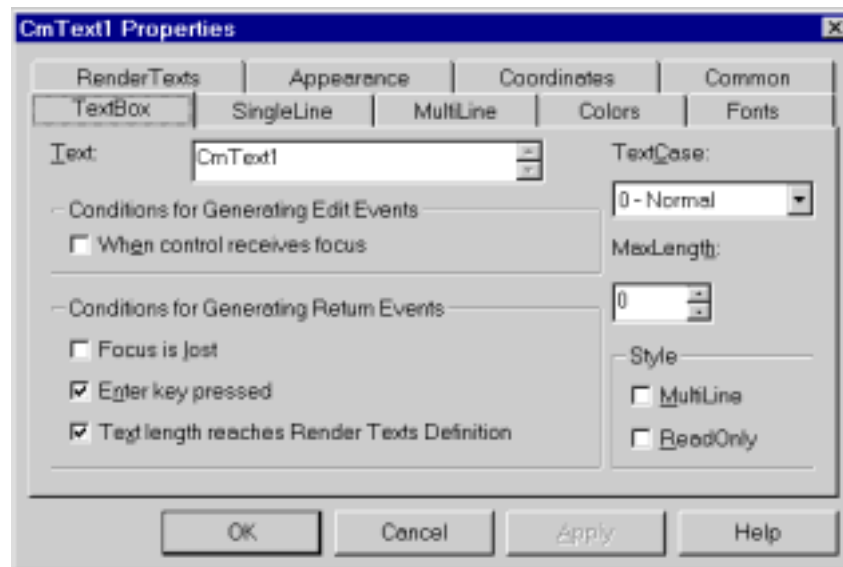


Figure 8.16. The Textbox Properties dialog box

Move the mouse to the Text: field and delete "CmText1" (this is the default value that appears in this field when the form is displayed).

Select the Common tab and change the control's name from CmText1 to EMPLOYEE-ID.

Click on the RenderTexts tab. Select the RenderStyle pulldown and click on 1 - COBOL PICTURE. This specifies that you want to enter a standard COBOL PICTURE STRING definition to enforce a particular format when the user enters data. In this case you will limit the size of the EmployeeNumber field to 9 (to match the size of the actual database key field to which it is associated) and ensure that the data entered is numeric.

Move the mouse to the PictureString field and enter:

9(9)

This defines a 9 character numeric field, and will prevent a user from entering more than 9 characters in this field when using this application. Click on the OK button to close the dialog box.

Note that when using the RenderTexts facility to define a numeric input, we first cleared the default text from the Text property. If you forget to do this, you will

receive an error when you attempt to apply a numeric value in RenderTexts. The reason for this is that RenderTexts immediately tries to apply the numeric field validation to the default text field (e.g. "CmText1"), which is obviously an invalid numeric value.

For the following additional two textbox controls you are about to place on the form, limit their size to 20 characters using the same option to match the corresponding database-defined field sizes.

Repeat the above process twice to create two additional textbox controls directly below the one you just created. Remember to set the field data type to X(20) using the RenderTexts tab option in the TextBox Properties dialog box for these two textbox controls.

For the second textbox control, change the name of the control to "FIRSTNAME" (located in the Common tab) and delete the default text value as well. Remember to limit the size of the field to 20 characters using the RenderTexts tab in the Properties dialog box.

For the third textbox control, change the name of the control to "LASTNAME" and delete the default text value as well. Remember to limit the size of the field to 20 characters.

When you are finished defining the three textbox controls, your form should look something like the following:

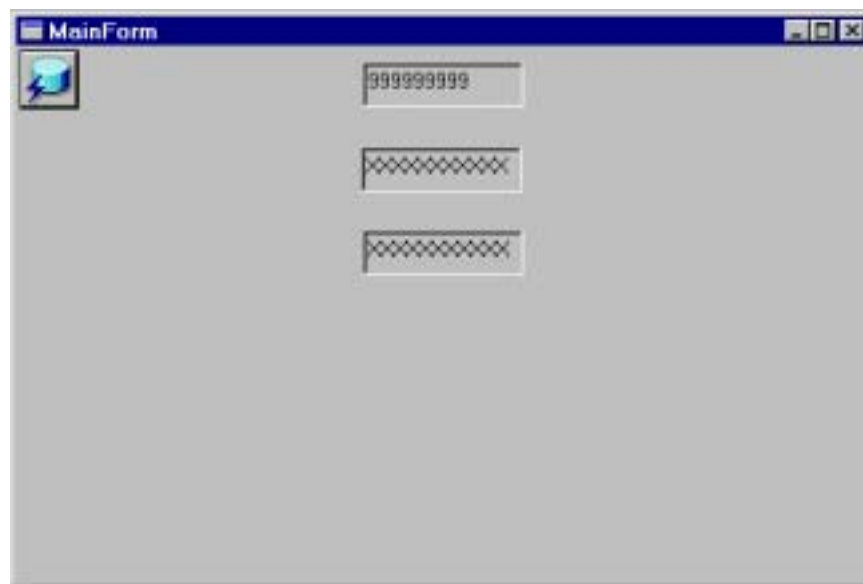


Figure 8.17. The form after defining the 3 textbox controls

Creating the Label fields

The next task is to place three label fields on the form to label the three textbox controls. When you are finished with this task, the form should appear as follows:

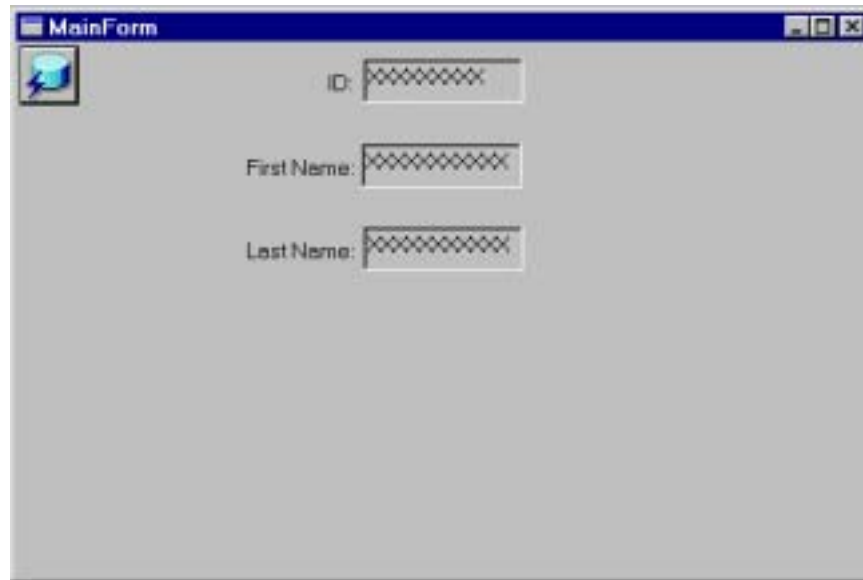



Figure 8.18. The form with three label fields added

If you have not placed the textbox controls in the proper location on the form, you can reposition them easily by left clicking the mouse on them and dragging them to a new location.

You may also left-click on the form outside of the textbox controls and draw a box around all three controls. Once you have all three controls contained within a box, you can select Center Controls from the Layout menu, and select the Horizontal option. This will cause all three controls to be centered horizontally on the form.

Now find the StaticText control  in the Toolbox palette and click on it with the left mouse button. This control is used to create labels on a form.

Move the mouse back to the form and drop the first label field to the left of the top most textbox control.

Right-click the mouse on this new label and select Properties from the pop-up menu (or highlight the label by clicking on it and select Properties from the Edit menu).


You do not need change the control's name, as it will not be referred to in the application - it is a static piece of text on the form that will not change. Change the default text value in the Caption field to "ID:".

To properly align the label, click on the Right option under Alignment (Horizontal), and then click on the Vert. Center option under Alignment (Vertical). Click on the OK button to close the dialog box.

Repeat this process twice more to create a middle label whose caption reads "First Name:", and a bottom label whose caption reads "Last Name:" to the left of the remaining two textbox controls (see figure 8.18 above).

Creating the Command Buttons

The next task is to define the two command buttons. (Refer to the example in Figure 8.15, which shows the completed form.)

Find the CommandButton control icon  in the Toolbox palette and click on it with the left mouse button. Move the mouse back to the form and 'drop' the first command button near the upper right corner of the form.

Right-click the mouse on it and select Properties from the pop-up menu.

Change the control's caption from "CmCommand1" to "Retrieve" and the control's name value from "CmCommand1" to "PUSH-RETRIEVE" (under the Common tab).

Repeat the above process to place a second CommandButton control near the lower right corner of the sheet. Change the control's caption to "Close" and the control's name value to "PUSH-CLOSE" (under the Common tab).

The form should now look like the example shown in Figure 8.15. You have now completed developing the graphical user interface for the application. The next step is to write the required event procedures to tie the application together.

Close the Form Editor by selecting Close from the File menu. Return to the PowerCOBOL Project Manager window, which should now appear something like the following:

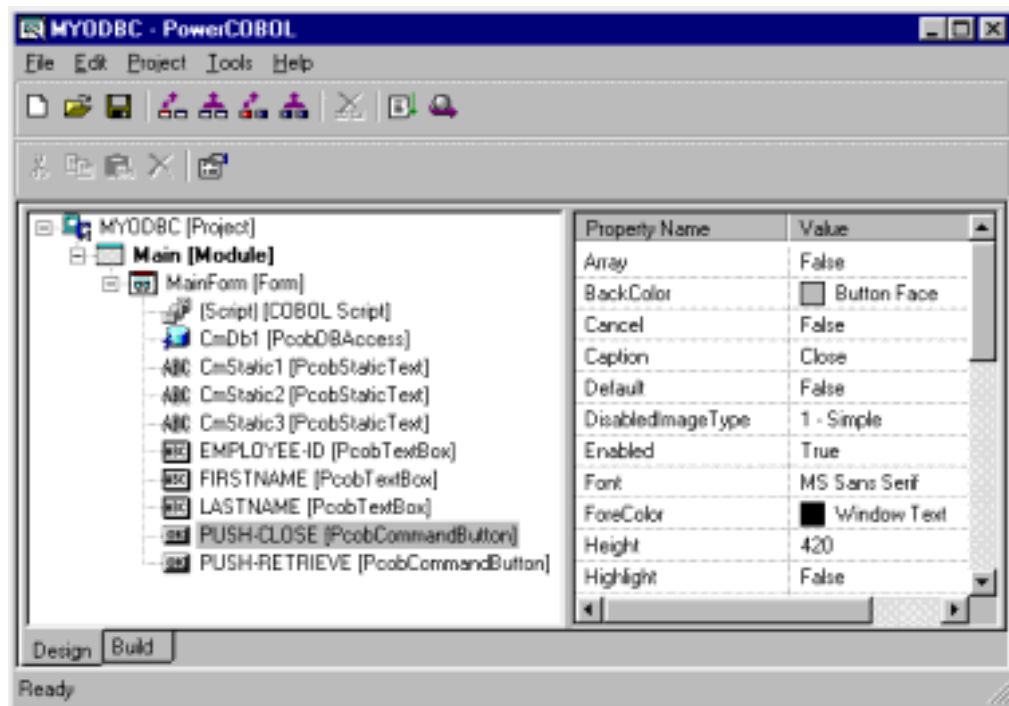


Figure 8.19. The Project Manager window after defining the GUI interface

Writing the Event Procedures

The next step is to create the event procedures needed to complete the application logic.

Hopefully, as you complete this process, you will be quite impressed at how little time you will spend developing such an application.

The application will make use of one global variable to monitor the open/close state of the database.

Instead of placing a third command button on the window and writing code for it to open the database, you will use the global variable and some clever programming logic to place the database open function in the "Retrieve" command button's event procedure.

The logic you are going to write will use the single Retrieve command button. When it is initially clicked, it will sense that the database is not open and will open it. It will also perform a "SelectRecords" function call to select all of the database records.

The logic will then perform a "ReadNextRecord" until it reaches the end of the database table unless you change the value of the EMPLOYEE-ID field. If you change the value of the EMPLOYEE-ID field, it will attempt a specific "SelectRecords" function using the value you enter.

Instead of looking for an exact match when doing a specific "SelectRecords" function on a given EMPLOYEE-ID value, the application will use the greater than or equal (\geq) operator to find the next match.

As complicated as this may sound, you are going to accomplish this using only a few lines of procedural code in PowerCOBOL.

You will first define the global data item needed to keep track of the current database open/close status. You are going to use a global variable so that you will be able to interrogate and set it from both of the separate event procedures you will define later.

Remember that data items defined in an event procedure are local data items to that procedure and may not be accessed directly from other event procedures.

You will thus define a global data item at the form level, which will allow any of the form's event procedures to access it directly and thus share it.

You will also define a second global data item (ReturnValue) that will be used by all ODBC calls to place a return value into, indicating success or an error code.

Make sure that you are currently viewing the application from the Project Manager window as shown in Figure 8.19.

Move the mouse to the MainForm form (the application's window) and right-click on it. From the pop-up menu that appears, move the mouse over Edit DATA DIVISION to bring up a secondary pop-up menu. Select WORKING-STORAGE.

This will bring up the PowerCOBOL Editor. Because you have not previously defined any WORKING-STORAGE items for the form, the Editor window will be empty. Enter the following lines of code in the Editor window and then save it and close the Editor:

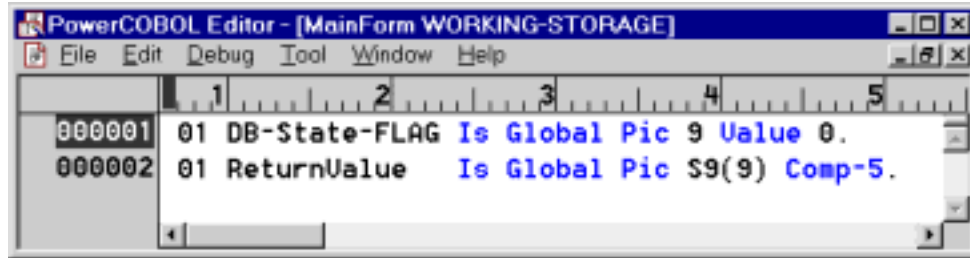


Figure 8.20. Creating global data items in the form's WORKING-STORAGE section

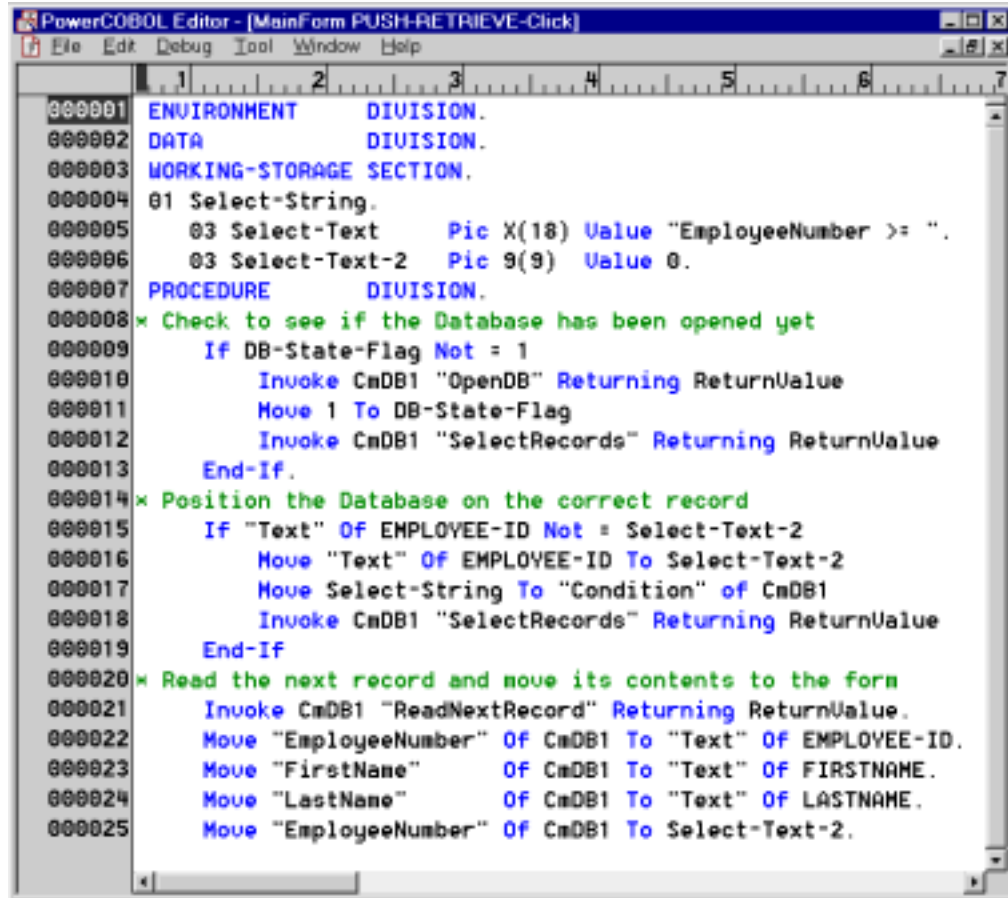
Now it's time to create the main logic portion of the code that will be associated with the Retrieve command button.

Note that the object name assigned to the DB Access control by PowerCOBOL is CmDB1. This control has been connected to the MYODBC ODBC data source name that you defined earlier.

Move the mouse to the PUSH-RETRIEVE command button definition in the left windowpane of the Project Manager window (you may need to expand the project to see all application components if the PUSH-RETRIEVE object is not displayed).

Right-click on it once with the mouse to display a pop-up menu. Move the mouse over Edit the event procedure and select the Click event. This tells PowerCOBOL that you wish to create an event procedure for the Click event (a click event takes place when the user moves the mouse over the command button and left-clicks on it).

This will bring up the PowerCOBOL Editor to allow you to enter the COBOL code associated with this event procedure. Enter the following code in the Editor window:



```

000001 ENVIRONMENT    DIVISION.
000002 DATA          DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Select-String.
000005     03 Select-Text    Pic X(18) Value "EmployeeNumber >= ".
000006     03 Select-Text-2  Pic 9(9) Value 0.
000007 PROCEDURE      DIVISION.
000008 * Check to see if the Database has been opened yet
000009     If DB-State-Flag Not = 1
000010         Invoke CnDB1 "OpenDB" Returning ReturnValue
000011         Move 1 To DB-State-Flag
000012         Invoke CnDB1 "SelectRecords" Returning ReturnValue
000013     End-If.
000014 * Position the Database on the correct record
000015     If "Text" OF EMPLOYEE-ID Not = Select-Text-2
000016         Move "Text" OF EMPLOYEE-ID To Select-Text-2
000017         Move Select-String To "Condition" of CnDB1
000018         Invoke CnDB1 "SelectRecords" Returning ReturnValue
000019     End-If
000020 * Read the next record and move its contents to the form
000021     Invoke CnDB1 "ReadNextRecord" Returning ReturnValue.
000022     Move "EmployeeNumber" OF CnDB1 To "Text" OF EMPLOYEE-ID.
000023     Move "FirstName"      OF CnDB1 To "Text" OF FIRSTNAME.
000024     Move "LastName"       OF CnDB1 To "Text" OF LASTNAME.
000025     Move "EmployeeNumber" OF CnDB1 To Select-Text-2.

```

Figure 8.21. The event procedure for the Retrieve command button

When you are finished, save the edit session and close it.

Now it's time to write the final code snippet for the Close command button. When the close button is selected, you need to close the database and exit the application.

In the Project Manager window, move the mouse to the Close command button definition and right-click on it to display a pop-up menu. Move the mouse over Edit the event procedure and select the Click event from the secondary pop-up menu that appears.

The PowerCOBOL Editor appears, ready for you to input the event procedure. Type in the code as shown in the following example:

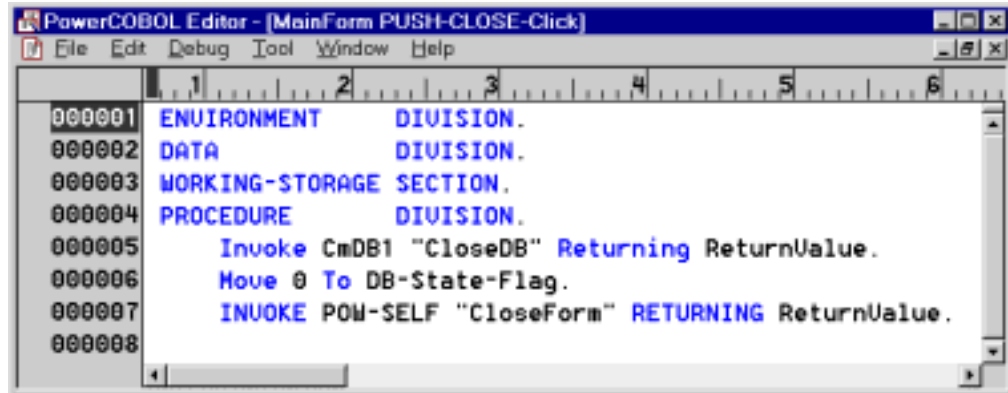


Figure 8.22. The event procedure for the Close command button

Notice the use of the control name "POW-SELF". In PowerCOBOL, POW-SELF contains the name of the current form. It is a convenient short cut to specifying the name of the current form. You could have optionally coded "MainForm" where POW-SELF appears. If you changed the name of "MainForm" later, however, you would be forced to change the name in any event procedure that references it. By using "POW-SELF", however, you need not worry if the name of the form changes.

Incidentally, if you would like to see a list of available database-related data fields (properties) and functions (methods), highlight the "CmDB1" string in the Editor window, and click on it once with the right mouse button.

From the pop-up menu that appears, you can select either Insert Method or Insert Property to see a list of either. If you select a property or method from one of these lists, PowerCOBOL will automatically build the appropriate procedural COBOL statement and place it into the Editor at the point of the cursor.

Save and close the edit session. You have completed the development portion of this project. That's all there is to it.

Select Save from the File menu to save the new additions to the project.

You are ready to build your application and run it.

Building and Running the Application

After saving your project, right-click the mouse on the project name in the Project Manager window and select All Build from the pop-up menu. If you receive any compile errors, return to the appropriate location to correct the errors.

Once you have executed a clean Build, you are ready to run your application for the first time. Right-click the mouse on the Main module and select Execute from the pop-up menu that appears.

The application should appear as follows:

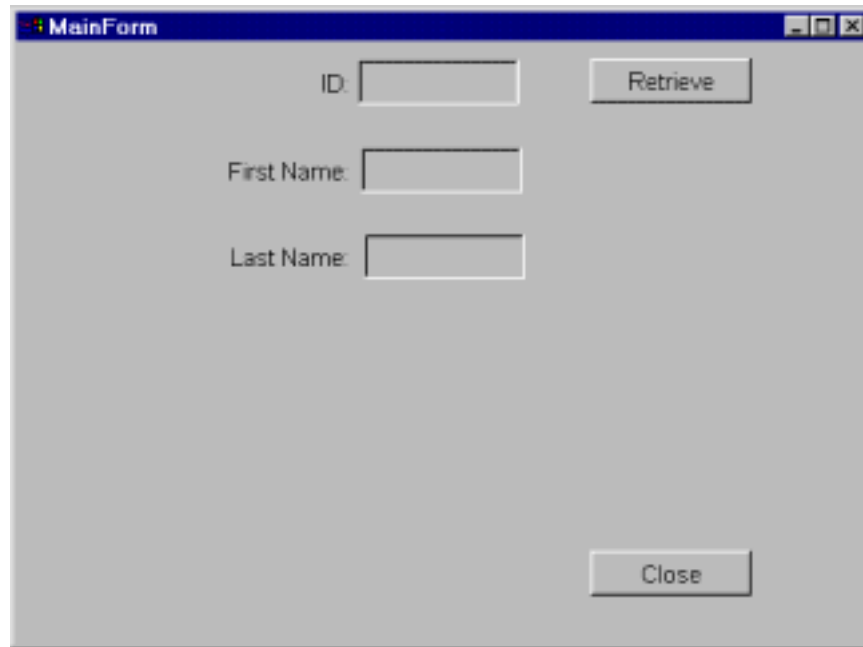


Figure 8.23. Executing the application for the first time

Click on the Retrieve button. This will cause the database to be opened and a record starting with the lowest possible key value of all zeroes to be selected.

The first record will be found and displayed in the window as shown in the following example:



Figure 8.24. The first record is displayed

Experiment with the application a bit. If you keep clicking on the Retrieve button you will move down the table (there are only 5 records in the table, incidentally). If you

enter an ID value of 000000001 through 000000005, you will retrieve that specific record. If you enter an invalid key, the prior value will be reset.

Notice that you can enter data in the First or Last Name fields but this data will be ignored if you click on the Retrieve button again.

When you are finished experimenting, close the application by clicking on the Close command button.

Enhancing the Application

You will now add some additional functionality to your application to allow you to add, delete, and update records in the table.

Make sure that you have closed the application and are back in the Form Editor on MainForm (move the mouse to MainForm in the Project Manager window, right-click the mouse on it and select Open from the pop-up menu).

You are now going to add three additional command buttons to the form. When you are finished, it will appear as follows:

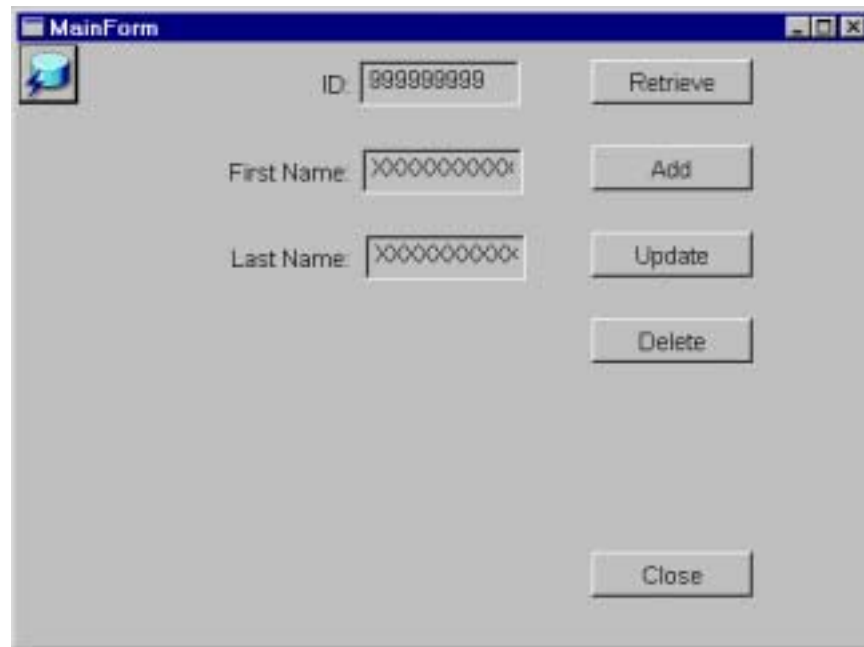


Figure 8.25. The form with three new command buttons added

Add the three command buttons (Add, Update, and Delete) as shown, by selecting the CommandButton Control icon in the Toolbox palette and dropping each button on the form in the appropriate location.

Open the Properties dialog box for each command button and change the caption to the appropriate button name shown in Figure 8.25. Additionally, change the command button object names to PUSH-ADD, PUSH-UPDATE, and PUSH-DELETE, respectively, by selecting the Common tab and typing the appropriate name in the name field.

Once these three new command buttons have been added, you are ready to write the event procedures to implement their intended function.

It is worth noting that the event procedure for all three new command buttons will be identical except for the second to last line in each event procedure, which calls the appropriate database function.

Bring up the Click Event Procedure for the Add command button in the Editor by right clicking the mouse on the Add button and selecting Edit the event procedure from the pop-up menu. Select the Click event from the secondary pop-up menu.

Type in the code as shown in the following example:

```

000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Temp-ID Pic S9(4) Comp-5.
000005 PROCEDURE DIVISION.
000006 * Check to see if Database is already open and open if not
000007 If DB-State-Flag Not = 1
000008 Invoke CnDB1 "OpenDB" Returning ReturnValue
000009 Move 1 To DB-State-Flag
000010 End-If.
000011 * The EmployeeID field is defined as Pic S9(4) Comp-5 in the
000012 * Database and the 9 character display numeric field on the
000013 * form must be converted to this format before being written
000014 * to the Database.
000015 Move "Text" Of EMPLOYEE-ID To Temp-ID.
000016 Move Temp-ID To "EmployeeNumber" Of CnDB1.
000017 Move "Text" Of FIRSTNAME To "FirstName" Of CnDB1.
000018 Move "Text" Of LASTNAME To "LastName" Of CnDB1.
000019 Invoke CnDB1 "WriteRecord" Returning ReturnValue.

```

Figure 8.26. The event procedure for the Add command button

Since the event procedure code for the next two command buttons is almost identical, you will copy this code and then modify it in the new event procedures.

Save the edit session for the Add button, but do not close the Editor.

Move the mouse back to the MainForm form and right-click on the Update command button. Move the mouse to Edit the event procedure and select the Click event from the secondary pop-up menu.

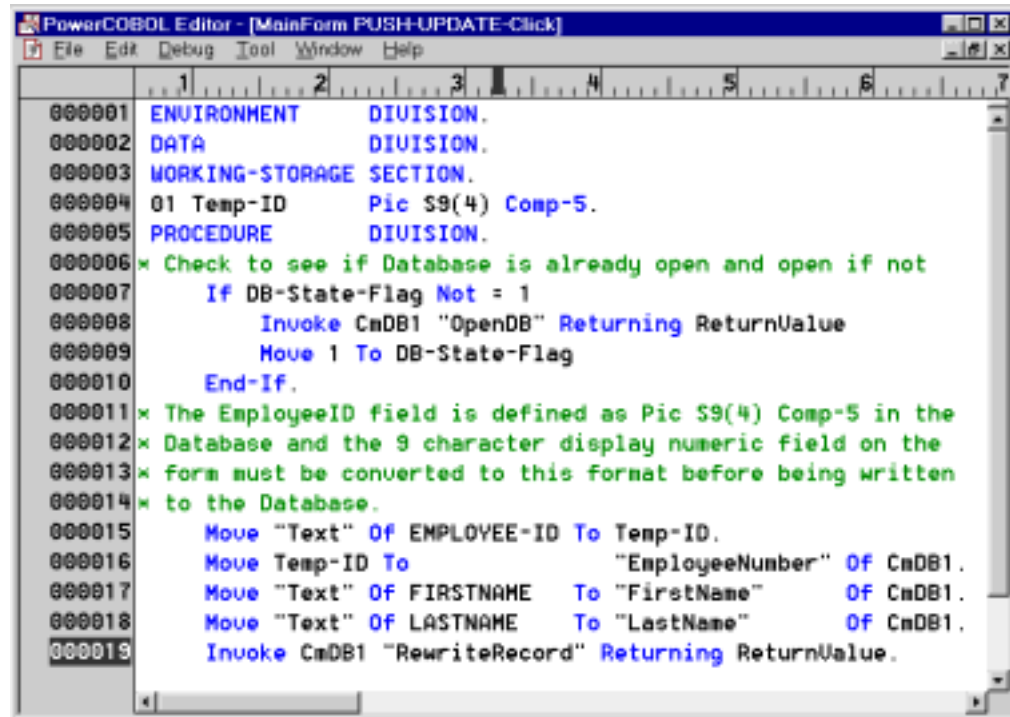
This brings up an Editor window for the Click event on the Update command button. In this new edit session, delete all of the code, by selecting Select All and then Delete from the Edit menu.

You should now have a completely blank Editor window. Now move the mouse back to the previously opened Click event procedure for the Add button.

Make a copy of this window's entire contents by selecting Select All and then Copy from the Edit menu. This places a copy of the entire event procedure onto the clipboard.

Now move the mouse back to the empty Editor window for the Click event for the Update button and click within the top left of the empty window edit area to place the cursor there.

Select Paste from the Edit menu, and the entire event procedure is now copied into this window. You need now only change a single line of code in this new event procedure. You need to invoke the RewriteRecord procedure instead of the WriteRecord procedure of CmDB1 on Line 17. Make this change as follows:



```

PowerCOBOL Editor - [MainForm PUSH-UPDATE-Click]
File Edit Debug Tool Window Help
1 2 3 4 5 6 7
000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Temp-ID Pic S9(4) Comp-5.
000005 PROCEDURE DIVISION.
000006 * Check to see if Database is already open and open if not
000007 If DB-State-Flag Not = 1
000008 Invoke CmDB1 "OpenDB" Returning ReturnValue
000009 Move 1 To DB-State-Flag
000010 End-If.
000011 * The EmployeeID field is defined as Pic S9(4) Comp-5 in the
000012 * Database and the 9 character display numeric field on the
000013 * form must be converted to this format before being written
000014 * to the Database.
000015 Move "Text" Of EMPLOYEE-ID To Temp-ID.
000016 Move Temp-ID To "EmployeeNumber" Of CmDB1.
000017 Move "Text" Of FIRSTNAME To "FirstName" Of CmDB1.
000018 Move "Text" Of LASTNAME To "LastName" Of CmDB1.
000019 Invoke CmDB1 "RewriteRecord" Returning ReturnValue.

```

Figure 8.27. The event procedure for the Update command button

Save and close the edit session for the Update command button, but leave the edit session for the Add command button open, as you will copy the code one more time for the Delete command button's Click event.

Repeat this same process for the Delete command button. Bring up an Editor window for the Delete command button's Click event, delete all of the code, and copy the code from the Add command button's Click event procedure.

Now change Line 17 from:

```
Invoke CmDB1 "WriteRecord" RETURNING ReturnValue
```

to:

```
Invoke CmDB1 "DeleteRecord" RETURNING ReturnValue
```

The code for the Delete command button should appear as follows:

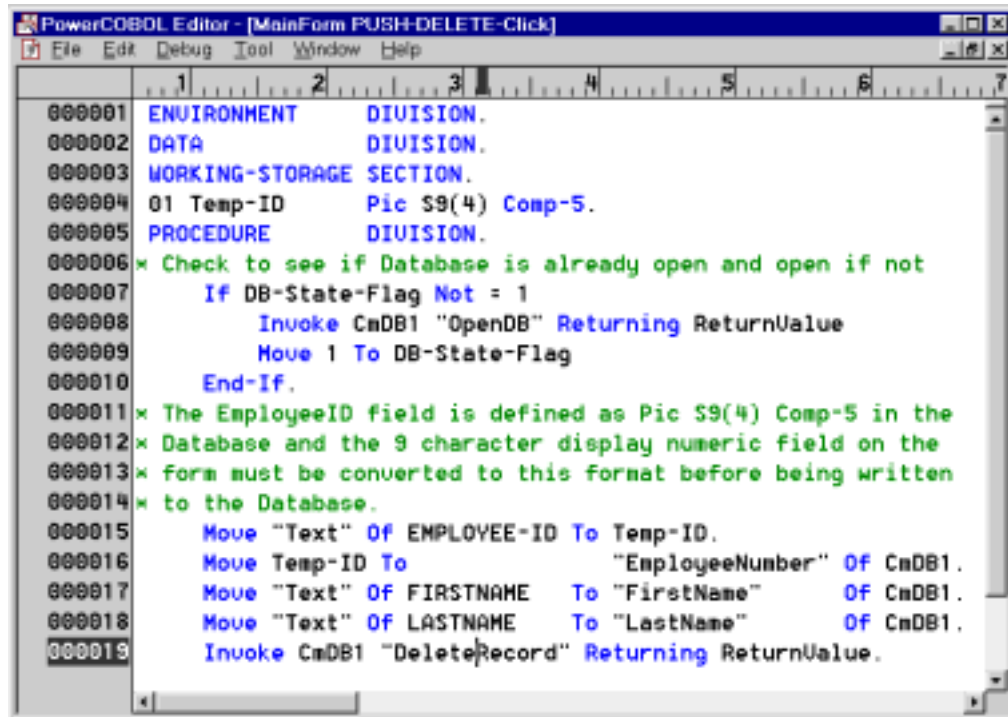


Figure 8.28. The event procedure for the Delete command button

Save and close the open edit sessions for the Add and Delete command buttons.

You have completed the enhancement process. Notice how relatively straightforward and simple it was to add a significant amount of new functionality to your application using PowerCOBOL!

Save your work (close the Form Editor and select Save from the File menu).

Right-click the mouse on the project name in the Project Manager window and select All Rebuild to rebuild the application.

If you receive any compile errors, go to the appropriate place(s) to fix them, save the project and click on All Rebuild option again.

When you have a clean build, click on the Run button (or select Run from the Project menu) and re-execute the application.

Experiment around a bit. You will find that you can now retrieve, update, add, and delete records in the table. You may even find PowerCOBOL performing some error checking for your application that you did not have to program in.

For example, try to add a duplicate record, and you should receive the following error dialog:

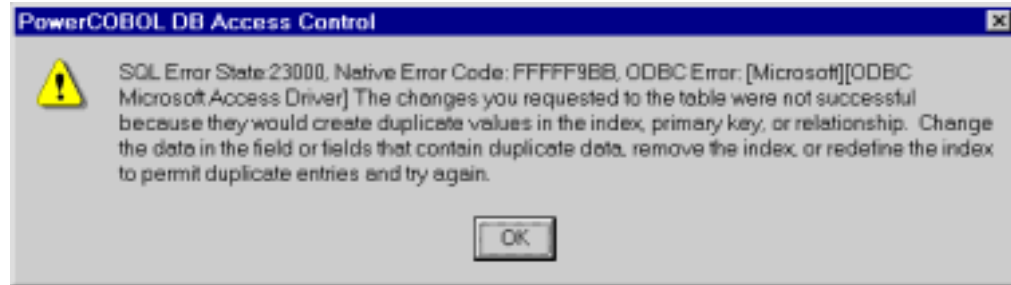


Figure 8.29. A PowerCOBOL error message on add duplicate key attempt

Note that the application you have just built does not have any error checking logic in it. You can easily implement this by examining the `ReturnValue` data item after each call (`Invoke`) to a database function.

You should now have a good understanding as to how to write an ODBC application in PowerCOBOL.

For more sophisticated applications, embedded SQL may be used with systems such as Microsoft's `SQLServer` to create client/server ODBC applications.

Chapter 9. PowerCOBOL Programming Techniques

This chapter discusses PowerCOBOL programming techniques and includes the following topics:

- Traditional COBOL programming methodology versus PowerCOBOL's event-driven programming methodology
- PowerCOBOL application execution flow
- Sharing data between PowerCOBOL forms and procedures
- Calling COBOL DLLs
- PowerCOBOL overall application architecture
- Working with multiple windows in a PowerCOBOL application
- Working with objects (using the *COM Class)
- Receiving object references in event procedures
- Opening a PowerCOBOL form from COBOL
- Calling the Windows API from PowerCOBOL
- Sharing data between a COBOL application and a PowerCOBOL form
- Working with VT_BSTR and VT_VARIANT Data Types
- Notes on programs that include COBOL and PowerCOBOL procedures

Traditional COBOL programming methodology versus PowerCOBOL's event-driven programming methodology

PowerCOBOL uses a different programming paradigm from that of conventional COBOL programming. The following sections discuss the differences.

Traditional COBOL Programming Methodology

Over the years, the COBOL language has evolved as a result of the various standards committees in place. COBOL is typically a *procedural* language (although the standards committee has approved object-oriented extensions to COBOL).

This means that traditional COBOL programs are structured to define files and data statically, followed by fixed procedures, which are executed in a structured, hierarchical pattern. The flow of control is strictly under the control of the programmer, and he or she is responsible for every aspect of the entire application.

To interact with terminal console screens, early COBOL provided the ACCEPT and DISPLAY verbs. These were and still are single line-oriented read and write functions for the console.

Over the years, COBOL screen I/O has evolved in a variety of ways from specific language vendors implementing proprietary extensions.

Some of these extensions have included enhancements to the ACCEPT and DISPLAY verbs to allow for full screen I/O, such as positioning on a specific row and column on the screen.

The implementation of the SCREEN SECTION by several vendors allowed for an entire screen layout to be predefined, complete with a variety of field attributes available. The ACCEPT and DISPLAY verbs were then used for an entire screen, as opposed to a single line on the display.

Vendors such as IBM implemented complex on-line transaction processors (OLTP) such as CICS and IMS/DC on mainframes.

This approach separated the screen handling from the program, defining an application programming interface for interacting with predefined screens. These screens were defined by writing assembler language macros (for example, BMS and MFS) which were compiled and assembled outside of the COBOL program.

The COBOL program was given a data and attribute definition block to work with, while all of the complexities of managing the actual screen I/O and physical display were taken care of by the On-line Transaction Processor (OLTP).

When graphical user interfaces (GUI's) first arrived, they required complex application programming interfaces. This proved to be a very complex and tedious development task for the typical programmer. It was common for 60% or more of a program's application logic to be dedicated to handling the GUI alone.

These programs proved to be extraordinarily complex, unstable, and non-portable to other GUI environments.

Realizing these problems, language tools vendors began developing higher level "GUI painter" products that would allow developers to create GUI components (for example, screens with push buttons) and would then generate large portions of the associated program code.

This approach proved helpful, but developers were still responsible for creating the entire application architecture and making sure all of the application components interacted properly.

Later, development tools vendors attempted to provide higher level GUI development tools that would allow the painting of an entire GUI for an application. This typically included a scripting language that allowed the developer to define the interaction between GUI components within the GUI development environment, and thus outside of the actual COBOL application.

The COBOL application could then make simple calls to the GUI run-time provided which would then manage the GUI outside of the actual application.

The main benefits from this approach were that the COBOL application code shrank significantly and programmers could concentrate on dealing with application-specific needs like file and database I/O.

Some of these GUI run-times were portable to other operating systems, and the overall GUI development process was simplified as it was logically separated from the application.

The drawbacks to this approach for COBOL developers were that the scripting languages were not in COBOL, and the developer was thus forced to learn a new language. The developer additionally had to be careful to design the application architecture to properly interact with the GUI run-time facility.

Another less obvious drawback was the fact that GUI's are typically object oriented and deal with the operating system in a recursive and object-oriented fashion, while COBOL has remained primarily a procedural language. COBOL developers are thus often forced to fit the procedural model of a COBOL program onto the more object-oriented model of a GUI.

PowerCOBOL solves these problems in the following ways:

PowerCOBOL Application Execution Flow

The components of PowerCOBOL, resources such as bitmaps, icons, cursors, and source files are referenced in a project file (.PPJ file). The output from a project is one or more executable modules (.EXE and/or .DLL files).

The following figure illustrates the components of a PowerCOBOL application including reference numbers tied to additional information on the following page.

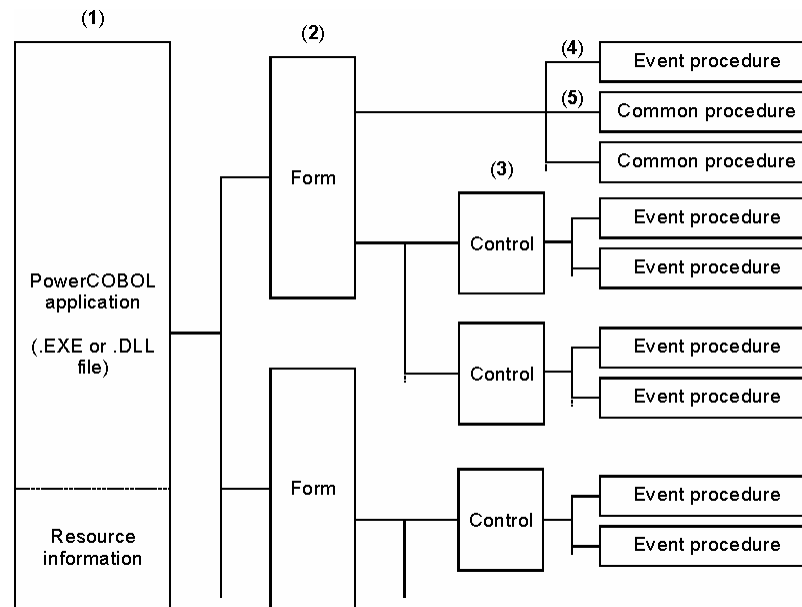


Figure 9.1. A PowerCOBOL application

1. **PowerCOBOL Application**
An executable program component (.EXE or .DLL file) that consists of one or more forms.
2. **Form**
An application window containing graphical controls that PowerCOBOL displays for you. PowerCOBOL treats each form as a COBOL compile unit.
3. **Control**
Graphical user interface (GUI) objects that are pasted onto a form. Controls may manipulate data in a PowerCOBOL application. At run-time, user actions with controls prompt events to be raised and associated event procedures to be executed.
4. **Event Procedure**
PowerCOBOL treats event procedures as unique, internal programs within a form. These procedures dictate the actions performed by the program in response to specified events.
5. **Common Procedure**
PowerCOBOL treats common procedures as standard, internal programs. These procedures describe processes that can be called from any event procedures in the form, and are thus sharable.

Event-Driven Programming Methodology

PowerCOBOL is based on an event-driven programming paradigm. In this paradigm, the program is constantly testing for and responding to a set of events, which are simple user actions such as mouse button clicks or keystrokes. PowerCOBOL programs use forms (windows) and graphical controls (objects) to create the graphical interfaces that interact with users.

A more detailed discussion of event-driven programming may be found in Chapter 1, "Introduction."

Executing a PowerCOBOL Application

The PowerCOBOL run-time system is required to run PowerCOBOL applications. It acts as an intermediary between the Windows operating system and the PowerCOBOL application by handling all the messaging and call formatting required to work within the Windows environment. The following figure illustrates the execution process used in PowerCOBOL applications.

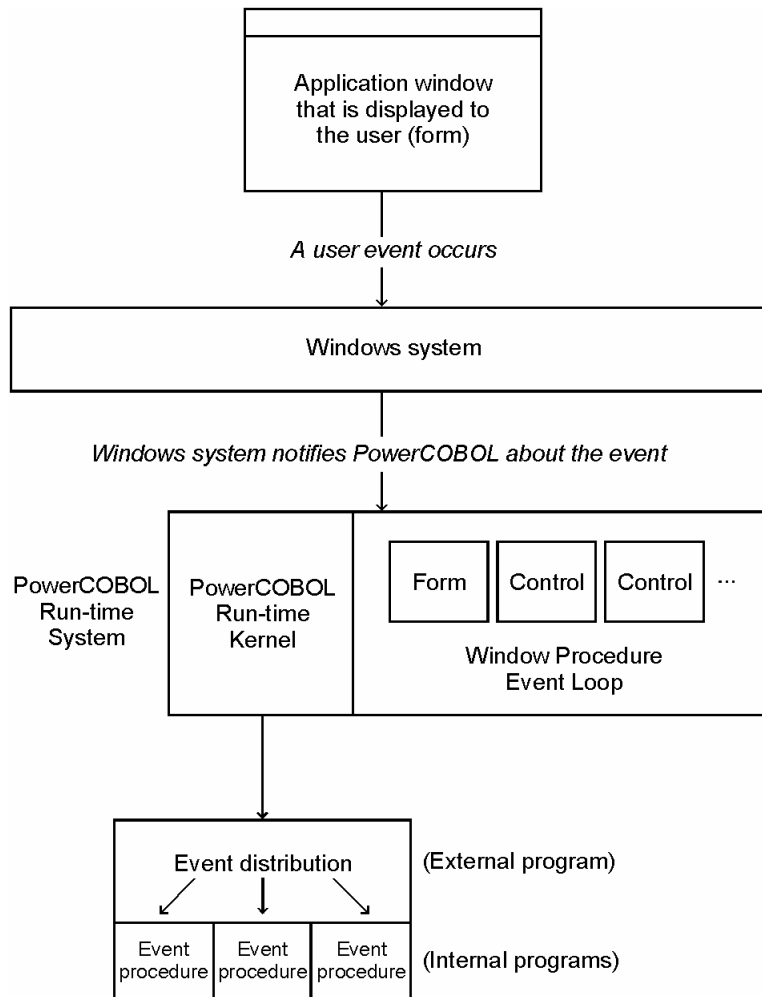


Figure 9.2. Executing a PowerCOBOL application

Sharing Data between PowerCOBOL Forms and Procedures

Within a PowerCOBOL procedure, you can typically only refer to data that is defined within that procedure.

If you wish to share data between individual procedures within a PowerCOBOL form, you must place the data declaration in the form's WORKING-STORAGE section, and specify the IS GLOBAL clause.

You do this by right clicking on the form name in the PowerCOBOL Project Manager and selecting Edit DATA DIVISION from the pop-up menu, then select WORKING-STORAGE from the secondary pop-up menu.

You may then insert a data declaration such as follows:

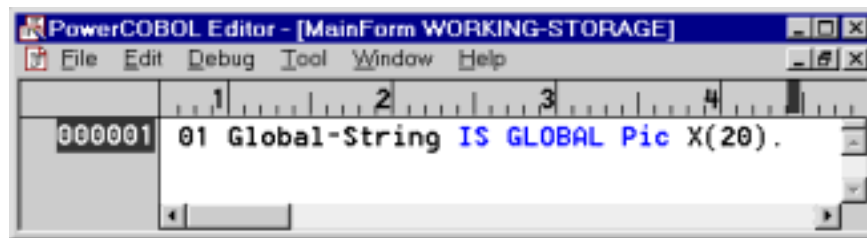


Figure 9.6. A Global data declaration in the form's WORKING-STORAGE section

If you wish to share data between separate forms and/or .DLL modules, simply add the EXTERNAL clause to the data declaration as follows:

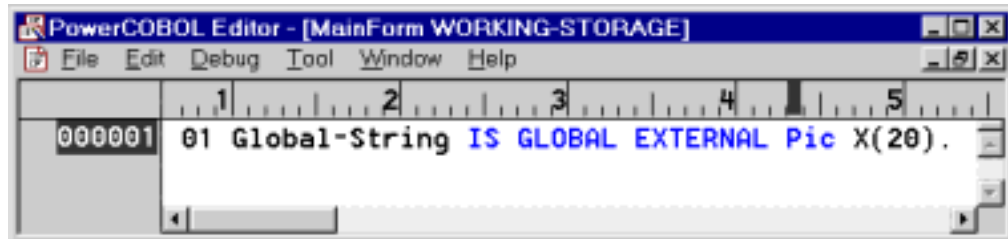


Figure 9.7. A Global External data declaration in the form's WORKING-STORAGE section

Note that anywhere you wish to reference a global or external data item, the current module must contain the identical definition in its Working-Storage Section.

And, global or external data can be used only when the Script Language property of a module is "0-COBOL85 Language Mode". If you wish to share data when the Script Language property is "0-OOCOBOL Language Mode", create an ActiveX control and use the custom properties and custom methods to refer/set the data.

Defining and Using Data Files in PowerCOBOL

Using Data files in COBOL is done using the same techniques that have been available in COBOL for many years. Within the Form Environment Division's FILE-CONTROL Section, you code COBOL SELECT statements to select a file, assign it to an external name, and define the type of file to be used such as:

```
Select INFILE Assign To Myfile
      Organization is Sequential
      File Status is WS-File-Status.
```

In a Form Data Division's FILE Section, you code FD statements such as:

```
FD INFILE GLOBAL.
01 INFILE-RECORD PIC X(132).
```

You then use standard COBOL OPEN, READ, WRITE, CLOSE, etc. statements to access the file.

You must, however, associate the filename ("MyFile" in the select statement above) with an external filename on disk). There are three separate techniques available to accomplish this:

1. You can hard code a PC file name in the actual Select clause such as:

```
Select INFILE Assign to "c:\data\myfile.dat"
```

2. If you do not place quotes around the file name in the Select clause, you can create a data item in Working-Storage with the same name and move a PC file name into it such as:

```
Select INFILE Assign to WS-File-Name
.
.
Working-Storage Section.
01 WS-File-Name      Pic X(30) Value "c:\data\myfile.dat".
```

3. You can assign to a file name without quotes and instead of declaring the file name in the Working-Storage Section as in item 2 above, you can declare this dynamically at runtime by placing an entry for the module name in a COBOL85.CBR file located in the same directory as the module. For example, if the name of the module being created is "Myprog.exe", you would place the following lines of code in a COBOL85.CBR file located in the same directory as the Myprog.exe module (remember that PowerCOBOL places the Myproj.exe file in a separate subdirectory under the directory containing the project file by default):

```
[MYPROG]
WS-File-Name=c:\data\myfile.dat
```

You are then free to use normal COBOL file I/O statements in your PowerCOBOL application. By adding the GLOBAL keyword to the FD clause, you may access the file from any event procedure in the same form. One event procedure may open the

file, while a second event procedure reads the file, while a third event procedure writes to the file, or one event procedure may do all three.

If you add the EXTERNAL keyword to a file's FD statement, and make the identical declaration in another form or even within a separate module, you may even share the file across forms or modules.

Line Sequential Data Files

NetCOBOL supports a special file format found in the PC environment known as LINE SEQUENTIAL. If you define a file as being LINE SEQUENTIAL in a COBOL SELECT statement, this means that each record written to the file will have all trailing spaces deleted and two bytes will be added as an end of record delimiter (Hex value X"0D0A", which represents a carriage control and line feed character).

If you are creating a file where records may have a great number of trailing spaces, this can conserve a great deal of physical space on disk. In fact, most word processors and editors store files in this format, which is sometimes called text format.

You can define a LINE SEQUENTIAL file for input or output by using the ORGANIZATION IS LINE SEQUENTIAL phrase in the file's SELECT clause.

Be careful, however that if you choose to use this format, you do not declare any non-display data types such as COMP, COMP-3 or COMP-5. The reason for this is that X"0D0A" is a valid COMP or COMP-5 value and if you are unlucky enough to have this value in a data file defined as LINE SEQUENTIAL, it will be mistakenly taken as an end of record and your record will be truncated at that point. Also note that X"09" is a tab character and can cause some undesired results as well.

A good rule of thumb is to only use the LINE SEQUENTIAL format for data files containing all PIC X and PIC 9 fields.

Note that if you code the following:

```
Select INFILE Assign to Myfile
      Organization is Sequential.
```

Your file will be RECORD SEQUENTIAL (no X"0D0A" will be added), which means it will be fixed length and may contain mixed data types.

Calling COBOL DLLs

Call COBOL Dynamic Link Library (DLL) files by specifying the "CALL identifier" syntax from within PowerCOBOL event procedures. To load data dynamically upon execution of the application, call the COBOL DLL file using a CALL statement. Import libraries are not necessary.

Before starting the program, create a module entry for any environment information required by the DLL in a COBOL85.CBR file and save this file in the same directory as the .EXE file which calls the .DLL file.

What is a DLL File?

A dynamic link library file, or a .DLL file, contains code and data that can be shared by several applications simultaneously. DLL files are bound to a program dynamically when it is loaded or executed, as opposed to statically when it is linked. In other words, the object module in the DLL file dynamically links to the object module in the program at run-time.

PowerCOBOL supports the development of DLL files. In COBOL programs, most of the procedures that are shared between applications are DLL files. Because the basic programming unit used in PowerCOBOL is a form, forms that are shared between applications become DLL files. A PowerCOBOL .DLL file may contain multiple forms as well.

In order to register .DLLs from within the Windows operating system, use the Register utility. You can access this utility by right clicking the mouse on a .DLL module in the Project Manager window and selecting Register from the pop-up menu that appears.

The Execution Environment

Generally, only one execution environment is built when a PowerCOBOL application is executed. However, when DLL files are used, several execution environments may exist within a single application.

Forms developed using PowerCOBOL can be called from either a PowerCOBOL application or a COBOL application. The PowerCOBOL run-time system is required when using PowerCOBOL constructs, such as forms. When you are running an application that has been developed exclusively with PowerCOBOL, the run-time environment is automatically available. This is not the case when calling a form from a COBOL program.

PowerCOBOL Overall Application Architecture

Understanding the overall architecture of a PowerCOBOL application is the key to developing consistent and efficient applications. In this section you will learn not only about application architecture, but also how to access files and data items across multiple forms and programs.

PowerCOBOL makes extensive use of newer enhancements to the COBOL language in order to effectively design the application architecture.

For many years, a major complaint about COBOL versus newer programming languages was that COBOL is too static.

Some other programming languages support the creation of functions that have their own local data items. In a traditional COBOL program, all data items must be predefined in the WORKING-STORAGE or LINKAGE Sections.

This means that these data items are accessible from any portion of the COBOL program.

Local data is the concept of having a function (for example, a sub-routine) that has its own data items, which are not made available to the rest of the application. Only the function is aware of its data and thus only the function may change its data.

This is one of the basic building blocks of object-oriented programming. It allows developers to break up complex applications into smaller, more easily managed functions that in turn encapsulate their data and methods.

The ANSI 85 specification thus includes support for embedded programs. This allows COBOL programmers to create independent COBOL programs (complete with a unique Program-ID) and to embed these independent programs right into an existing COBOL program. They may then be called like normal external COBOL programs.

Because these programs are embedded into the main application, they are much more efficient to communicate with at run-time. Additionally, they may encapsulate their own data and sub- functions (methods) within an application.

NetCOBOL fully supports embedded programs. PowerCOBOL takes advantage of this by creating embedded programs for each event procedure and/or form procedure created by a developer.

The most recent enhancement to the COBOL language is object- oriented extensions.

PowerCOBOL can create ActiveX controls (created as .DLL files) and register them with the Windows operating system.

The following figure shows the overall architecture of a PowerCOBOL application.

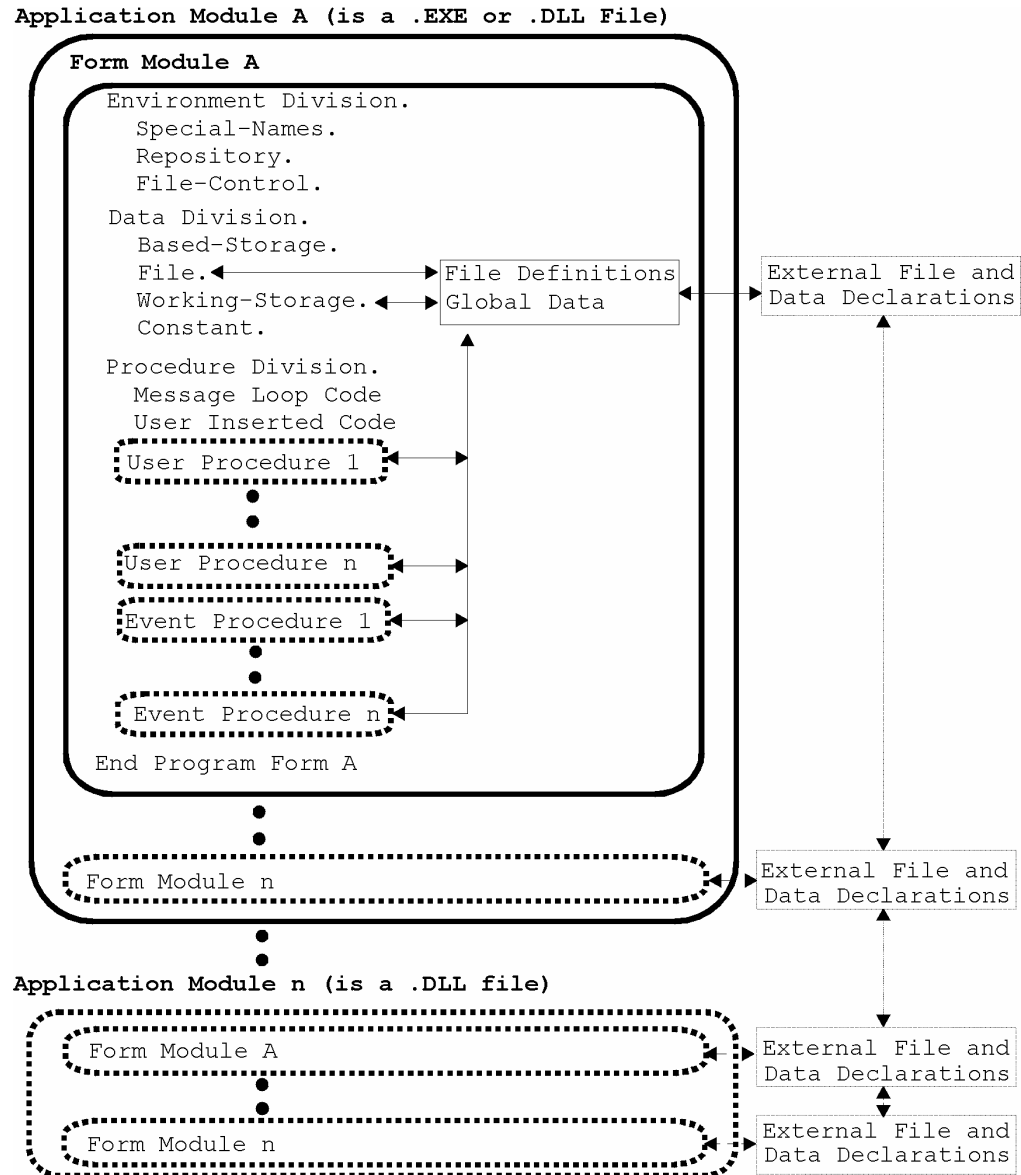


Figure 9.8. The overall architecture of a PowerCOBOL application

In reviewing Figure 9.8 above, the following facts should be considered:

- A PowerCOBOL application may consist of one or more executable application modules. The main application module may be either an .EXE or a .DLL file. All other application modules should be .DLL files. Each of these executables may contain one or more COBOL programs.
- Each form that is defined in a PowerCOBOL application will cause a separate COBOL program module to be created. A PowerCOBOL executable application module may contain one or more form modules.
- Each form module will have one general area for its file and data declarations, along with constants and other COBOL specific declarations at the form level.
- Each form module will contain an event loop in its PROCEDURE DIVISION. Additionally, any number of user specified procedure statements may be added

to the form's PROCEDURE DIVISION. The event loop will exist as a COBOL EVALUATE statement. The PowerCOBOL run-time system will manage Windows events and pass them into the appropriate form module. The form module in turn evaluates each event passed into it and branches to the appropriate event procedure. This means that control comes into the form module from external events through the form's PROCEDURE DIVISION.

- Each form's PROCEDURE DIVISION may contain any number of user-specified procedures or event procedures. All of these procedures will physically exist as embedded COBOL programs. The IS COMMON clause is automatically added to each Procedure's PROGRAM-ID clause to allow it to be accessed from anywhere within the form module.
- Each form may specify one or more data items as global, by using the IS GLOBAL clause in the DATA DECLARATION, thus allowing these items to be accessed and shared by all of the user or event procedures defined within that specific form.
- Each form may specify one or more file definitions as global by using the IS GLOBAL clause on the FD statement in the form's DATA DIVISION File section.
- Form file definitions and data items may be shared across forms in the same executable application module by declaring them external by using the IS EXTERNAL clause. Any file definition or data item that you wish to share across forms must be identically declared in each form's FILE and/or WORKING-STORAGE sections.
- Form file definitions and data items may be shared across separate executable application modules by declaring them using the IS EXTERNAL clause as well. Identical file and/or data item definitions must exist in both application modules including the IS EXTERNAL clause.
- Individual user and event procedures contained within a form module (which themselves exist as embedded COBOL programs), may contain file definitions and data item definitions. Attempting to use the IS GLOBAL clause will not, however, allow these items to be shared outside of that specific user or event procedure.
- User and event procedures are static by default. This means that data items are not automatically re-initialized upon re-entry into one of the procedures. When a user or event procedure is exited, it is not canceled, thereby preserving the current state of the data when it is re-entered. If you wish to ensure that one of more data items are initialized each and every time a specific user or event procedure is entered, use the COBOL INITIALIZE verb for each data item required.
- A form module may invoke another form module within the same executable application module or even within a separate executable application module. In both cases the OPENFORM or CALLFORM method is called.
- Note that an application .EXE module may only share data with a .DLL module. An .EXE cannot call another .EXE and share data using external data declarations.
- Note that it is potentially dangerous to add raw procedural code to a form's PROCEDURE DIVISION unless you select New under the Edit PROCEDURE DIVISION pop-up menu. Using the New option forces any code you enter to be encapsulated into an embedded COBOL program complete with a unique

Program-ID (which will become the user procedure name). This is illustrated in figure 9.8 above as a User Procedure. One potential use of the form's Procedure option is to specify a #INCLUDE statement to copy in some number of embedded programs which may exist outside of the application. Make sure that each of these programs contained in the #INCLUDE file contains a unique Program-ID and an END PROGRAM statement at the end of each.

Working with Multiple Windows in a PowerCOBOL Application

Invoking Sub-Windows

There are two ways of invoking child forms: using the OpenForm method and using the CallForm method.

- OpenForm opens the child form as a modeless window i.e. you can interact with the parent window as well as the child window.
- CallForm brings up the child form as a modal window i.e. you cannot interact with the parent window until the child window is closed. You use modal windows when displaying dialog boxes, in which you require user input before you can proceed with the next function.

The methods are invoked in identical ways, apart from the method name. For example, to open SUBFORM1 contained in the same executable you code:

```
INVOKE POW-SELF "OpenForm" USING "SUBFORM1".
```

or

```
INVOKE POW-SELF "CallForm" USING "SUBDLG1".
```

To open SUBFORM2, contained in another DLL, SUB.DLL for example, you code:

```
INVOKE POW-SELF "OpenForm" USING "SUBFORM2" "SUB.DLL".
```

or

```
INVOKE POW-SELF "CallForm" USING "SUBDLG2" "SUB.DLL".
```

Passing Data between Forms

To pass data from one form to another use EXTERNAL data items. To use EXTERNAL data items, define the same item in the WORKING-STORAGE SECTIONS of both parent and child forms.

For example, if you want to pass an 8 byte item called FORM-DATA from the parent to the child form, you would code the following in the WORKING-STORAGE SECTIONS of both forms:

```
01 FORM-DATA IS GLOBAL EXTERNAL PIC X(8).
```

Returning a Value from a Child Form opened by CallForm

For modal child forms, i.e. those opened using the CallForm method, you can also pass a data value to the parent form by using the result parameter in the CloseForm method. This value is set in the RETURNING parameter of the CallForm statement.

For example, suppose you want to take different actions in the parent form depending on whether the OK or Cancel button was used to close the child form. In the Click event of the child form's OK button you would code:

```
WORKING-STORAGE SECTION.

01 ID-OK PIC S9(9) COMP-5 VALUE 1.

PROCEDURE DIVISION.

    INVOKE POW-SELF "CloseForm" USING ID-OK.
```

And in the Click event of the Cancel button you would code:

```
WORKING-STORAGE SECTION.

01 ID-CANCEL PIC S9(9) COMP-5 VALUE 0.

PROCEDURE DIVISION.

    INVOKE POW-SELF "CloseForm" USING ID-CANCEL.
```

Then, in the parent form, the code to invoke the child form (SUBDLG1) and handle the different return values would look like this:

```
WORKING-STORAGE SECTION.

01 RETURN-VALUE PIC S9(9) COMP-5.

PROCEDURE DIVISION.

    INVOKE POW-SELF "CallForm" USING "SUBDLG1" RETURNING RETURN-VALUE.

    IF RETURN-VALUE = 1 THEN
*>        Process the OK button being clicked

    ELSE
*>        Process the Cancel button being clicked

    END-IF
```

Identifying which Form has been Closed

When using OpenForm to invoke child forms you can have several child forms open at one time. When a child form is closed you may want to take a specific action in the parent form using the CloseChild event, so you need to know which form has been closed. You do this by passing each child an ID (called a "cookie") when you use the OpenForm (or CallForm) methods. The ID of the closed form is passed as a parameter to the CloseChild event.

For example, suppose you are going to invoke two child forms "SUBFORM1", contained in the same DLL as the parent form, and "SUBFORM2" contained in a different DLL. You would first define ID's for the child forms in the parent form:

```
01 SUBFORM1-ID PIC S9(9) COMP-5 VALUE 1 IS GLOBAL.
01 SUBFORM2-ID PIC S9(9) COMP-5 VALUE 2 IS GLOBAL.
```

You then use these ID's when you open the subforms:

```
INVOKE POW-SELF "OpenForm" USING "SUBFORM1" SUBFORM1-ID.
INVOKE POW-SELF "OpenForm" USING "SUBFORM2" "SUB.DLL" SUBFORM2-ID.
```

In the CloseChild event of the parent form you can then code (POW-COOKIE is defined for you automatically by PowerCOBOL):

```
LINKAGE SECTION.
01 POW-COOKIE PIC S9(9) COMP-5.
PROCEDURE DIVISION USING POW-COOKIE.
    EVALUATE POW-COOKIE
    WHEN SUBFORM1-ID
*>        Process SUBFORM1 being closed

    WHEN SUBFORM1-ID
*>        Process SUBFORM2 being closed

    END-EVALUATE
```

Working with Objects

PowerCOBOL applications are essentially combinations of objects, where objects are collections of properties, methods and events with particular behaviors.

Controls and forms are special types of objects but the PowerCOBOL design is such that you never need to refer to controls or forms as objects. Because the Form is not a control on the control bar, the PowerCOBOL reference describes a Form as an object, otherwise using and referencing forms is much like using and referencing controls.

The other items described as objects - Button, Column, Font, ListItem and Node - can be referenced as objects of classes. This section describes how you do this. When this section refers to PowerCOBOL objects it means one of the above objects, excluding the Form object.

Accessing Objects using PowerCOBOL Syntax

Objects can be accessed using the PowerCOBOL syntax - as described in the PowerCOBOL Reference. The format is similar to accessing controls except that the object is always qualified by the control of which it is a part.

Objects are either referenced directly or, when there can be several occurrences of the object within a control, through a property that is an array of pointers to the objects.

For example to set the Text property of the fourth ListItem object in the Listview1 control you would code:

```
MOVE "This is the fourth item" TO "Text" OF "ListItems" (4) OF ListView1
```

This format is convenient for simple applications and uses, but in some situations it is more convenient to use object references rather than the object name or pointer arrays. Using object references is described in the following sections.

Accessing Objects Using Class Object References

In certain situations you may want the same code to access different objects, or you may find it more convenient to refer to an object reference rather than the fully qualified PowerCOBOL syntax format (e.g. "Child" (n) OF "Child" (m) OF "Root" (p) OF TreeView1). To enable you to do this PowerCOBOL provides classes for the objects as listed in Table 2.1 below.

Table 2.1. PowerCOBOL Classes for PowerCOBOL Objects

Object	Class Name
Column	POW-CCOLUMN
Font	POW-CFONT
ListItem	POW-CLISTITEM
Node	POW-CNODE
Others	POW-COBJECT

To use one of these classes you set up a USAGE OBJECT REFERENCE item in WORKING-STORAGE and MOVE the object to that item. For example, to set up a data item NODE1 to store the object reference for a TreeView node you could code:

```
01 NODE1 USAGE IS OBJECT REFERENCE POW-CNODE.
...
MOVE "Root" (2) OF TreeView1 TO NODE1
```

You can use the object reference, NODE1, wherever you would use the fuller syntax "Root" (2) OF TreeView1'. For example, you can set properties, or reference child nodes:

```
MOVE "NODE-NAME" TO "Text" OF NODE1.
MOVE "Child" (3) OF NODE1 TO NODE2.
```

See the TreeView sample program for examples of using object references.

Accessing Objects Using the COBOL *COM Class

NetCOBOL provides a *COM class for working with automation servers. This class provides methods for creating objects, invoking the object methods and accessing the object's properties.

The *COM class can be useful for accessing objects created by other systems where there is no matching POW-???? class.

To access PowerCOBOL objects or controls using the *COM class:

1. Right click on an unused part of the Form, and select Edit ENVIRONMENT DIVISION from the pop-up menu.
2. From the sub-menu select REPOSITORY.
PowerCOBOL opens an edit window for you to type the entries for the REPOSITORY paragraph.
3. In this window type:

```
CLASS COM AS "*COM"
```

And close the REPOSITORY window.

4. In like manner, from the form, select Edit DATA DIVISION, WORKING-STORAGE, and type:

```
01 COMNODE USAGE IS OBJECT REFERENCE COM.
```

5. Finally, to get an object reference of the appropriate class in COMNODE, you use a call to the routine "POWERCONVTOCOM" (Convert reference to an COM class reference) in your event code.

```
CALL "POWERCONVTOCOM" USING NODE1 RETURNING COMNODE.
```

You can then use the COMNODE item with the functions provided for the *COM class.

NOTE: You should clear COMNODE when you are finished using it by coding:

```
SET COMNODE TO NULL.
```

Using the *COM Class to access Microsoft ADO

The following example shows a PowerCOBOL application using the *COM class to access Microsoft's ADO (Active Data Objects).

It uses an ADO Connection object to establish a connection to the database, and uses an ADO Recordset object to read records back from the database and place them into a ListBox control on the form.

You must remember when using the *COM class to define it in the form's REPOSITORY by right clicking on the form, selecting Edit ENVIRONMENT DIVISION, and then clicking on REPOSITORY. Then place the following line of code in the REPOSITORY section and save it:

```
CLASS COM AS "*COM".
```

In one of the form's event procedures, the following code will open an Access data base, read the records and add them to a ListBox control named LST-DISPLAY:

```

ENVIRONMENT      DIVISION.
DATA             DIVISION.
WORKING-STORAGE SECTION.
01 OBJ-CONNECTION OBJECT REFERENCE COM.
01 OBJ-RECORDSET  OBJECT REFERENCE COM.
01 OBJ-FIELDS     OBJECT REFERENCE COM.
01 OBJ-FIELD      OBJECT REFERENCE COM.
01 PROGID-CONNECTION PIC X(8192) VALUE "ADODB.Connection".
01 PROGID-RECORDSET  PIC X(8192) VALUE "ADODB.Recordset".
01 NAME-PROVIDER     PIC X(8192) VALUE "Microsoft.Jet.OLEDB.4.0".
01 NAME-DB           PIC X(8192) VALUE "c:\sample.mdb".
01 NAME-TABLE        PIC X(8192) VALUE "Employee".
01 NAME-FIELD        PIC X(8192) VALUE "Name".
01 WK-VAL            PIC X(8192).
01 WK-0             PIC S9(9) COMP-5 VALUE 0.
01 WK-1             PIC S9(9) COMP-5 VALUE 1.
01 IS-EOF           PIC S9(4) COMP-5.

PROCEDURE        DIVISION.
* Create ADO Connection object and Connect to DB
  invoke COM "CREATE-OBJECT" using PROGID-CONNECTION
    returning OBJ-CONNECTION.
  invoke OBJ-CONNECTION "SET-Provider" using NAME-PROVIDER.
  invoke OBJ-CONNECTION "Open" using NAME-DB.

* Create ADO Recordset object and associate with "Employee" table
  invoke COM "CREATE-OBJECT" using PROGID-RECORDSET
    returning OBJ-RECORDSET.
  invoke OBJ-RECORDSET "Open" using NAME-TABLE
    OBJ-CONNECTION
    WK-0
    WK-1
    WK-0.

* Copy "Name" field of "Employee" table to ListBox
  invoke OBJ-RECORDSET "GET-EOF" returning IS-EOF.
  perform with test before until IS-EOF not = 0
    invoke OBJ-RECORDSET "GET-Fields" returning OBJ-FIELDS
    invoke OBJ-FIELDS "GET-Item" using NAME-FIELD
      returning OBJ-FIELD
    invoke OBJ-FIELD "GET-Value" returning WK-VAL
    invoke LST-DISPLAY "AddString" using WK-VAL
    set OBJ-FIELD to Null
    set OBJ-FIELDS to Null
    invoke OBJ-RECORDSET "MoveNext"
    invoke OBJ-RECORDSET "GET-EOF" returning IS-EOF
  end-perform.

* Clean up
  invoke OBJ-RECORDSET "Close".
  set OBJ-RECORDSET to Null.
  invoke OBJ-CONNECTION "Close".
  set OBJ-CONNECTION to Null.

```

Note that, from Version 6.1, PowerCOBOL ships with an ADO Data control to ease ADO development. This means that you do not need to use the *COM approach shown above to access ADO, but this example has been left in this manual to illustrate one of the more complex examples of using the *COM class.

Receiving Object References in Event Procedures

Some events receive object references as parameters. This section explains two ways of receiving those parameters:

- using appropriate class object references

- using *COM class object references.

Receiving Object References Using Class Names

If you want to use class object references, as described in “Accessing Objects Using Class Object References” above, you need to receive the object reference in a USAGE OBJECT REFERENCE item of the appropriate class.

Event procedures that receive object references usually have skeleton object references inserted, by PowerCOBOL, in the LINKAGE SECTION of the event code. For example, in the NodeClick event of the TreeView control, the following is displayed by default in the LINKAGE SECTION:

```
01 POW-PCMNODE OBJECT REFERENCE [Class-name].
```

For a node object, the appropriate class is POW-CNODE so you overwrite “[Class-name]” with “POW-CNODE”, giving:

```
01 POW-PCMNODE OBJECT REFERENCE POW-CNODE.
```

You can then use POW-PCMNODE to access properties and invoke methods.

Receiving Objects Using the COBOL *COM Class

When you work with ActiveX controls created by other systems you need to handle object references using the *COM class. For convenience, the example below uses a PowerCOBOL object, but you will be using external ActiveX controls.

To handle the objects passed to events using the *COM class, you need to declare the class in the REPOSITORY paragraph of the ENVIRONMENT DIVISION as described in “Accessing Objects Using the COBOL *COM Class” above.

Then, in the event code insert a “USAGE OBJECT REFERENCE COM” item in the WORKING-STORAGE SECTION and delete “[Class-name]” from the default LINKAGE SECTION entry. For example, in the NodeClick event you would have:

```
WORKING-STORAGE SECTION.
01 COMPCMNODE OBJECT REFERENCE COM.
LINKAGE SECTION.
01 POW-PCMNODE OBJECT REFERENCE.
```

To get a COM class object reference you convert the received object reference using the “POWERCONVTOCOM” sub-program:

```
CALL "POWERCONVTOCOM" USING POW-PCMNODE RETURNING COMPCMNODE.
```

Note: You should clear COMNODE after you finish using the statement:

```
SET COMNODE TO NULL.
```

Opening a PowerCOBOL Form from COBOL

This section explains how to open a form developed in PowerCOBOL from a standard (non-PowerCOBOL) COBOL program. To open the form, it must be compiled into a .DLL file.

Syntax

Call "POWEROPENFORM" USING DLL-Name Form-Name

Parameters

Parameter	Property	Meaning
DLL-Name	X(260)	The name of the DLL where the form exists.
Form-Name	X(14)	The name of the form.

NOTES

- The 'POWEROPENFORM' method will not return control to the calling program until the form has been closed.
- Ensure that you code the DLL-Name and Form-Name data items noted above using the correct size as indicated.
- A PROGRAM-STATUS of zeroes or a positive number indicates success. A negative value indicates failure.
- When you do not specify a path in the DLL-Name data item, the .DLL being called is searched for in the following order:
 - Current directory
 - Windows directory
 - Windows system directory
 - Directory where the executable program exists
 - Directories listed in the PATH environment variable
- Every form within an application must have a unique name. Do not attempt to open a form that is already open. PowerCOBOL will return an error message if you do.
- You cannot call the program recursively if the COBOL85 Language Mode is specified in this module's properties.
- The form is not closed until after the all of the event programs are finished. So, close the child forms before you close the parent form.
- If you use the NetCOBOL "*COM" Class you cannot use the "Activate" method. You should use the "DoModal" method.
- Refer to the PowerCOBOL sample programs OpenActiveX.ppj and CallActiveX.ppj for further information.

Life span of DLL's created by PowerCOBOL

A DLL created by PowerCOBOL is loaded when an included form in the DLL is opened, and the DLL is unloaded when all included forms are closed. After unloading, the values of all data items are cleared.

You should thus make an entry (COBOL sub program) in the DLL and link it into the application that calls the DLL when the application is loaded if you need to keep the DLL loaded between the application start and finish.

Refer to the COBOL User's Guide for details of dynamic structure linking.

Preventing Multi-execution of the application

The module level ExecuteInDuplicate property controls the behavior of a PowerCOBOL .exe file when a user attempts to launch it for execution a second time. There are three values available for this property. They are listed with their related results below:

- 0 - Multi Instance - allows multiple instances of the application to be launched.
- 1 - Single Instance (Message) - only allows a single instance of the application to be executed and displays an error dialog box when the user tries to launch the application a second time.
- 2 -Single Instance (Activate) - only allows a single instance of the application to be executed and focuses the user on the currently running instance if he/she attempts to launch a second instance.

Note that PowerCOBOL treats multiple execution of an application in the same path separately. Therefore, PowerCOBOL recognizes an application that resides in another folder as a different application even if they were built from the same project file. For example, the following are different applications:

- An application saved in the local path and same application saved in the network path.
- An application copied to another path.

The Execute in Duplicate property is effective when the FileType property of module is "0-Execute Module". Note that, the ExecuteInDuplicate property should be used to prevent multiple access to a file. In cases where the same data file may be accessed concurrently from multiple instances of the same application (or from different applications), you need to use record file and/or record locking. For example, to open a file for exclusive use and lock it from other currently running applications:

FILE-CONTROL

```
SELECT LOCKFILE
ASSIGN TO LF
FILE STATUS IS LF-STATUS
LOCK MODE IS EXCLUSIVE.
```

FILE

```
FD LOCKFILE GLOBAL.
01 OAREA PIC X.
```

WORKING-STORAGE

```
01 LS-STATUS PIC X(2) GLOBAL.
```

OPENED Event

```
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
OPEN OUTPUT LOCKFILE WITH LOCK
IF LS-STATUS NOT = "00" THEN
    INVOKE POW-SELF "DisplayMessage" USING "OPEN ERROR!"
    INVOKE POW-SELF "Deactivate"
END-IF.
```

Calling the Windows API from PowerCOBOL

You may call the Windows API directly from PowerCOBOL (as well as directly from NetCOBOL). You will need to link into your PowerCOBOL application the appropriate library. NetCOBOL ships with the main Windows API library (USER32.LIB).

In order to add this into your project for linkage purposes, you right click on the module name in the Project Manager and select the Insert File option from the context menu that appears. Then navigate to the location of the USER32.LIB file, which by default is installed in the C:\Program Files\Fujitsu NetCOBOL for Windows\COBOL subdirectory, and select it.

The following example will change your desktop's wallpaper setting (your desktop's background) to a new .bmp file that you specify.

And you need to specify the "ALPHAL(WORD)" compile option in the Script's properties dialog box.

The following code placed in an event procedure will bring up the Windows get filename dialog box and allow you to select a .bmp file. It will then call the Windows API to set the desktop wallpaper to the BMP file you specify:

```
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ReturnValue PIC S9(9) Comp-5 Value 0.
01 FileMask PIC X(32)
    Value "Bitmap Files|*.bmp|All Files|*.*".
01 DeskFile.
    03 FileName PIC X(255) Value Spaces.
    03 Filler PIC X Value X"00".
01 Sub1 PIC 9(4) Comp-5 Value 0.
01 uiAction PIC S9(9) Comp-5 Value 0.
01 uiParam PIC S9(9) Comp-5 Value 0.
01 pvParam PIC X(255) Value Spaces.
01 fWinIni PIC S9(9) Comp-5 Value 0.
01 SPIF_UPDATEINIFILE PIC S9(9) Value 1.
01 SPIF_SENDWININICHANGE PIC S9(9) Value 2.

PROCEDURE DIVISION.
* Note that a special compiler directive is being used. Right click
* on (Script)[COBOL Script] in the project manager.
* Select "Properties" from the context menu and check
* the ALPHAL(WORD) compiler directive.
* Get the name of the bitmap file to set desktop background to
    Move Spaces to FileName.
```

```

        INVOKE POW-SELF "GetFileName" USING FileName
                                     "Select Bitmap File"
                                     FileMask
                                     POW-CDOPE
        RETURNING ReturnValue.

* If ReturnValue is true, user did select a .bmp file to apply
  If ReturnValue = POW-TRUE

* String parameters being passed to "C" functions must be null
* terminated. Check the filename and insert a null at the end of
* the filename
    Compute Sub1 = Function Length(FileName)
    Perform With Test Before Until (FileName(Sub1:1)
                                   Not = Space)

        Subtract 1 From Sub1
    End-Perform
    If Sub1 < 255
        Add 1 To Sub1
        Move X"00" To FileName(Sub1:1)
    End-If

* Set up parameters to call the "C" Function
    Move DeskFile To pvParam
    Move 20       To uiAction
    Move 0        To uiParam
    Move 0        To fWinIni

* Set the SPIF_UPDATEINIFILE Flag
    Add SPIF_UPDATEINIFILE To fWinIni

* Set the SPIF_SENDWININICHANGE Flag
    Add SPIF_SENDWININICHANGE To fWinIni

* Call the User32.dll SystemParametersInfoA function
    Call "SystemParametersInfoA" With STDCALL Linkage
                                     Using By Value uiAction,
                                     By Value uiParam,
                                     By Reference pvParam,
                                     By Value fWinIni
        Returning ReturnValue

    If ReturnValue = 1
        Invoke POW-SELF "DisplayMessage"
            Using "Desktop Background Changed!"
    Else
        Invoke POW-SELF "DisplayMessage"
            Using "Error - Desktop Background Not Changed!"
    End-If.

```

Sharing Data between a COBOL Application Calling a PowerCOBOL Application

It is possible to pass data between a COBOL program that calls a PowerCOBOL application using the POWEROPENFORM method noted above.

This is done simply by defining the same data item name(s) in both the COBOL program and the PowerCOBOL application at the appropriate locations using the IS GLOBAL EXTERNAL clause.

This clause should immediately follow the actual data item name in its COBOL declaration as in the example below:

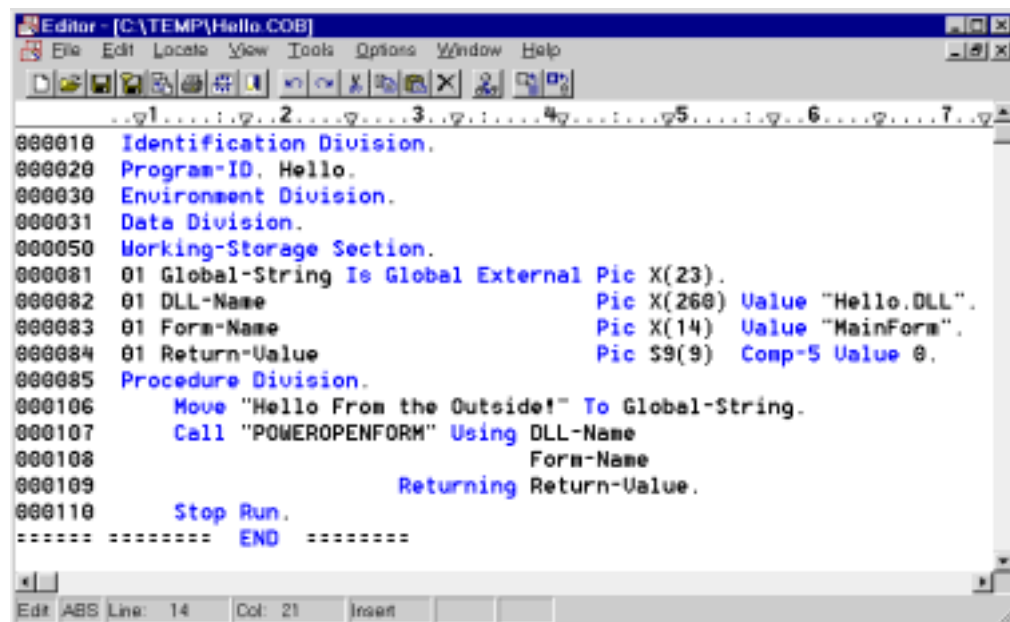
```
01 Global-String Is Global External Pic X(23)
```

You may include any number of such definitions in your application. Within the PowerCOBOL application, you may define this in the form's WORKING-STORAGE SECTION under its DATA DIVISION. To do this, right-click on the form name and select this from the pop-up menu, or on in the WORKING-STORAGE SECTION of any event procedure within the form.

Note that if you place a Global External data declaration with a specific procedure of a PowerCOBOL form, this data is only available to the PowerCOBOL application within that specific procedure - it cannot be accessed from other procedures in the same form. If you want to share this data between multiple procedures, you must declare it in the form's WORKING-STORAGE section.

Within a COBOL application, you define this in the application's standard WORKING-STORAGE section.

Below is an example of a sample COBOL application that calls a PowerCOBOL form named "MainForm" contained in a module built into a .DLL named "Hello.DLL":



```

Editor - [C:\TEMP\Hello.COB]
File Edit Locate View Tools Options Window Help
-----
000010 Identification Division.
000020 Program-ID, Hello.
000030 Environment Division.
000031 Data Division.
000050 Working-Storage Section.
000081 01 Global-String Is Global External Pic X(23).
000082 01 DLL-Name Pic X(260) Value "Hello.DLL".
000083 01 Form-Name Pic X(14) Value "MainForm".
000084 01 Return-Value Pic $9(9) Comp-5 Value 0.
000085 Procedure Division.
000106 Move "Hello From the Outside!" To Global-String.
000107 Call "POWEROPENFORM" Using DLL-Name
000108 Form-Name
000109 Returning Return-Value.
000110 Stop Run.
===== END =====
Edit ABS Line: 14 Col: 21 Insert

```

Figure 9.9. A sample COBOL program calling a PowerCOBOL form

Notice that this program contains a Global External data declaration.

The Click event procedure of the MainForm form in the PowerCOBOL application contains a similar declaration as follows:

```

000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Global-String Is Global External Pic X(23).
000005 PROCEDURE DIVISION.
000006 Move Global-String To "Text" Of CmText1.
000007

```

Figure 9.10. The Click event procedure for the PowerCOBOL form

When the COBOL program shown in figure 9.9 is executed, it calls the MainForm form in the PowerCOBOL application.

MainForm is then displayed. When MainForm's command button control is clicked on and the event procedure shown in figure 9.10 is executed, the Global-String data passed from the COBOL program is displayed on the form:

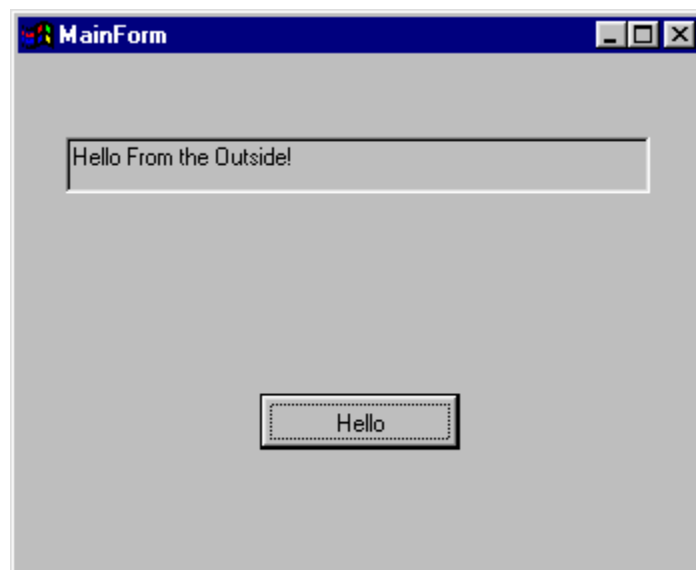


Figure 9.5. MainForm displayed with data from the COBOL application

Working with VT_BSTR and VT_VARIANT Data Types

Converting VT_BSTR data

PowerCOBOL supports VT_BSTR data used for properties or parameters of a method as follows:

PowerCOBOL converts VT_BSTR data items to alpha-numeric text by default. But it gets converted to another type depending on the target data type. When executing a MOVE statement, VT_BSTR data gets changed to the same type as the receiving data item. In ADD, SUBTRACT and COMPUTE statements, it is changed to a numeric data type. For example, a numeric data type can be moved to the Text property of a TextBox control, which is VT_BSTR data, as follows:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUM PIC S9(4) COMP-5 VALUE 0.
PROCEDURE DIVISION.
...
MOVE NUM TO "Text" OF CmText1
```

Note that it will not be converted between a property and another property. For example, the Style property of ComboBox control, which is VT_I2 data, cannot be moved to the Text property of a TextBox control.

```
MOVE "Style" OF CmCombo1 TO "Text" OF CmText1 => wrong result
```

In this case, you should move the property to an intermediate data item to perform the proper conversion as follows:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUM PIC S9(4) COMP-5 VALUE 0.
PROCEDURE DIVISION.
...
MOVE "Style" OF CmCombo1 TO NUM
MOVE NUM TO "Text" OF CmText1
```

Handling of spaces

PowerCOBOL handles space characters as follows:

All spaces at the end of a string are removed when the string is moved into a VT_BSTR data type. For example, if the following statement is executed, string " A" is displayed (" A " is not displayed).

```
MOVE " A " TO "Caption" OF CmStatic1
```

Operation for VT_VARIANT data

PowerCOBOL operates on VT_VARIANT data used for properties or parameters of a method as follows:

PowerCOBOL converts it to data types depending on the target data types similar to the way VT_BSTR is converted. The VT_VARIANT type conversion is not decided

until execution time, so there may be some cases that fail in converting. If a failure occurs, PowerCOBOL outputs an error message. In this case, you should check the program setting the VT_VARIANT data and change the value, which can be set, or use an intermediate Working-Storage data area as noted in the description of "Converting VT_BSTR data".

Note that controls and objects provided by PowerCOBOL do not offer the VT_VARIANT data type for the properties or parameters of their methods.

Notes on Programs that Include COBOL and PowerCOBOL Procedures

One of the benefits of PowerCOBOL is its support for traditional COBOL syntax. There is some complexity involved, however, when traditional COBOL (COBOL 97) and PowerCOBOL procedures are intermingled. This section describes some of the issues you need to be aware of, and provides examples of each.

1. Recursive calls are not permitted in COBOL.

Windows supports recursion. In other words, Windows allows programs to call and be called by each other. However, the COBOL standard does not support the recursive call of procedures. To avoid errors when using recursive procedures in PowerCOBOL, limit them to event procedures. Do not attempt to call COBOL procedures recursively from a PowerCOBOL event procedure.

2. The PowerCOBOL execution environment shuts down when you call a COBOL procedure from within a PowerCOBOL procedure.

If you open a PowerCOBOL form from within a COBOL program, a new PowerCOBOL execution environment is built. And if this COBOL program is called from within another PowerCOBOL event procedure, two execution environments are built.

Follow these two rules when using PowerCOBOL:

1. Each form name must be unique within an application.
2. Do not open the same form twice within an application.

NOTE: These rules do not extend across execution environments. For example, you cannot open a SUBFORM in a new execution environment if the SUBFORM has already been opened in the original execution environment. If you try to open the same form in the same execution environment within the same application, the run-time system returns an error code. The run-time system does not catch this error if the execution environments differ. For example, you can open SUBFORM from DLLFORM, but the new SUBFORM will destroy the area in which the old SUBFORM was stored.

3. A form will not close until all of the processing event procedures are completed.

NOTE: Even if the Deactivate method is issued, a form will not close until all of the event procedures in process come to completion.

Events process asynchronously. Generally there is little or no delay between the time that the Deactivate method is issued, and the actual closing of the form. Sometimes, if a long event procedure is processing when the Deactivate method is issued and control is returned to the Windows operating system, the form will be open until the procedure comes to completion.

Although a sequence of events takes place when you click on the Open button in the MAIN form, ultimately, the DLLFORM is opened. So, when you click on the Close button all of the procedures along the way, including the DLLFORM, must close before the MAIN form can close.

NOTE: The POWEROPENFORM method will not relinquish control of the application until every sub-procedure closes.

Chapter 10. Using PowerCOBOL Methods and Properties

This chapter explains how to use some of the methods and properties supplied with PowerCOBOL. You can think of methods as useful built-in functions or subprograms. Properties can generally be thought of as data items that contain some value important to the internal workings of a control.

You may additionally come across the use of the term *dialog* in describing methods. The Windows operating system provides several useful common dialogs to aid developers. Dialog is used to describe certain methods that allow user interaction, such as selecting a file name to open.

The terms *method* and *property* actually come from object-oriented programming terminology. In object-oriented programming, *objects* are collections of data encircled by the *methods* (functions) that act upon that data. The data is typically encapsulated and thus only accessible by first going through the object's methods. This prevents outside applications and methods from having direct access to the data, therefore ensuring that the data is only manipulated using the methods designed for it.

Properties are typically data items within an object that do not require a method to access them. Controls are typically designed to take some action when a particular property is changed. For example, simply moving a new color to a form's background color property (BackColor) will cause the form's background to change to the new color without being forced to call a method.

The Windows operating system graphical user interface (GUI) is highly object oriented and provides a number of methods and properties available at a low level to interact with it.

PowerCOBOL abstracts these lower level methods and properties and provides a set of higher-level methods and properties to make the building of GUI applications more straightforward to its users.

These methods and properties will vastly simplify and solidify your development efforts. For example, the GetFileName dialog will present the common Windows Open File dialog box and allow a user to browse drives and directories to find a file name to open. Writing the code for this method from scratch is both tedious and complex. Having the already well-tested method available makes it a snap.

Moving the value of False (provided as a constant in PowerCOBOL named POW-FALSE), to the Visible property available in many controls will make the entire control or even a complete form invisible. Move the value of True (POW-TRUE) to the same property will cause the control to become visible again.

This chapter will provide useful information on how to use some of these supplied methods and properties. The PowerCOBOL on-line help system contains information on every method and property available in PowerCOBOL. It also contains coding examples. If you are new to this type of programming you should spend some time examining these in the help system. PowerCOBOL additionally comes with many sample programs found by default in the Program Files\Fujitsu NetCOBOL for Windows\COBOL\SAMPLES\PowerCOBOL subdirectory. You should examine these projects to learn how to use controls and their supplied methods and properties if you need additional help.

Finding Method Information

Method Related Terminology

While browsing through the methods available in PowerCOBOL, it is useful to understand a few terms used in Windows-based development.

There are a number of methods that belong to functional groups associated with a specific technology area. In some cases, few if any methods exist for a particular technology area because it is so highly automated.

These include:

ActiveX – This is the latest rendition of Microsoft's OLE technology (discussed below) based upon Microsoft's COM (component object model). You can use PowerCOBOL to build ActiveX controls and components.

When you create an ActiveX component (control) using PowerCOBOL, its properties and methods are exposed to the outside world in a generic way, allowing it to be plugged into another application (which may or may not be written in PowerCOBOL), and manipulated like any other control or subprogram. ActiveX is a very powerful and exciting technology. Using it, you can embed a web browser control directly into your PowerCOBOL application and control it.

Think about how difficult it must be to write your own custom web browser. Using ActiveX, you simply grab the Microsoft Internet Explorer control and drop it onto your PowerCOBOL form. You may then access all of its properties and methods to customize it anyway you desire.

ADO - Active Data Objects - this is Microsoft's latest data access technology. It provides a more object-oriented way of accessing a variety of database systems in the Windows environment than the more traditional ODBC. PowerCOBOL version 6.1 provides an ADO data control.

DDE - Dynamic Data Exchange (DDE) is a facility that allows a Windows application to request information from another Windows application. DDE can sometimes be thought of as a kind of clipboard between separate applications at run-time.

DDE is a precursor to OLE, and OLE is generally considered to be a superior technology in many cases when compared to DDE. Technically, a DDE conversation takes place between two windows, but in reality most controls qualify as 'windows'. DDE is used when you establish a conversation (sometimes called a *link*) between two windows. One window becomes the *client* (the window requesting data), and the other window becomes the *server* (the window providing the requested data). The client can, however, send data back to the server. Windows will allow an application to engage in multiple DDE conversations at the same time. A single application may function as both a client and a server at the same time. You must know the DDE name of the application you wish to talk to in order to establish a link to it.

Two popular applications that support DDE are Microsoft Word and Excel, whose DDE names are "WinWord" and "Excel", respectively. You additionally need to know the *topic* of the DDE conversation. This is often simply a filename. Microsoft Excel will use a full path qualified filename ending in an .xls extension

as a suitable topic. Lastly, you need to know the *item* you are talking about. In using Excel as a DDE server, the DDE conversation item would typically be a cell or range of cells. If a PowerCOBOL application is the DDE server, the control name of a static text control, or image control, or a text box control may qualify as an item.

There are typically three types of conversations - *hot link*, *cold link*, and *notification link*. A hot link causes the server to send data contained in the specified item whenever it changes on the server in real time. A cold link means that the client must specifically request updates. A notification link is a conversation in which the DDE server alerts the DDE client that data within a specified item has changed, but the data is only transferred when a LinkRequest is executed.

MCI - Media Control Interface (MCI) is a technology that provides a number of multimedia-oriented controls such as a sound player and recorder, video player, CD player, etc. to your application. You can send commands to these devices to load, play, record, and save multimedia files. There are additional commands available that provide a high level of control over the MCI device being used. Using the PowerCOBOL MCI control, you can build your own CD player in COBOL, complete with buttons to control playing, rewinding and pausing. Sound a bit far fetched? If so, simply look at the PowerCOBOL sample project named CDPLAYER.PPJ found in the MCI subdirectory under the above noted sample programs directory.

ODBC - Open Data Base Connectivity (ODBC) is a technology that provides a common interface into a large number of database and file systems. Many database management systems (DBMS) come with a proprietary application programming interface (API) to program to. Some of these provide a relational SQL API, but there may be slight differences in the SQL usage that affect portability to other DBMS'.

ODBC was developed as a generic API to program to, which also allows the application to dynamically link to a database system using an ODBC source at run-time. This allows applications to easily switch between DBMS' without source code changes, as well not needing to be re-compiled or re-linked.

OLE - Object Linking and Embedding (OLE) began as a technology that complimented and extended dynamic data exchange (DDE). In later times it has been extended far beyond its initial uses.

One of the major differences between OLE and DDE is that in addition to transferring data between applications, OLE allows the data to appear in the same format as it does in the application sending the data. For example an Excel spreadsheet appears as an Excel spreadsheet when embedded in a PowerCOBOL form. You may even have access within the PowerCOBOL to the Excel tool bar and functions to act upon this embedded spreadsheet. Using OLE you are actually embedding a foreign application object directly into your PowerCOBOL application.

Additionally, *OLE Automation* allows a PowerCOBOL application to take control of another application and drive it. Using OLE automation, you can link to another OLE-enabled application and control its properties and even invoke its methods, thus controlling it completely.

Invoking Methods

Methods are accessed by using the COBOL INVOKE statement. The INVOKE verb is somewhat similar to a COBOL CALL statement, and was implemented as part of the object-oriented extensions to the COBOL language.

The general format of a method invocation is:

```
INVOKE Control-name "Method-name"
    [USING [Parameter-1]... ]
    [RETURNING [Return-Value]... ]
```

Note that method names must be contained in double quotes and each name is case sensitive, meaning it must be spelled exactly as documented in upper and lower case letters.

Determining Which Methods are Available and How to Use Them

The PowerCOBOL on-line help system provides an excellent resource for accessing information on available methods and how to use them, including information on parameter formats and values, along with return codes.

To access this information from anywhere within the PowerCOBOL development environment, simply move the mouse to the Help pulldown menu and select Help Topic.

The following dialog box will appear:

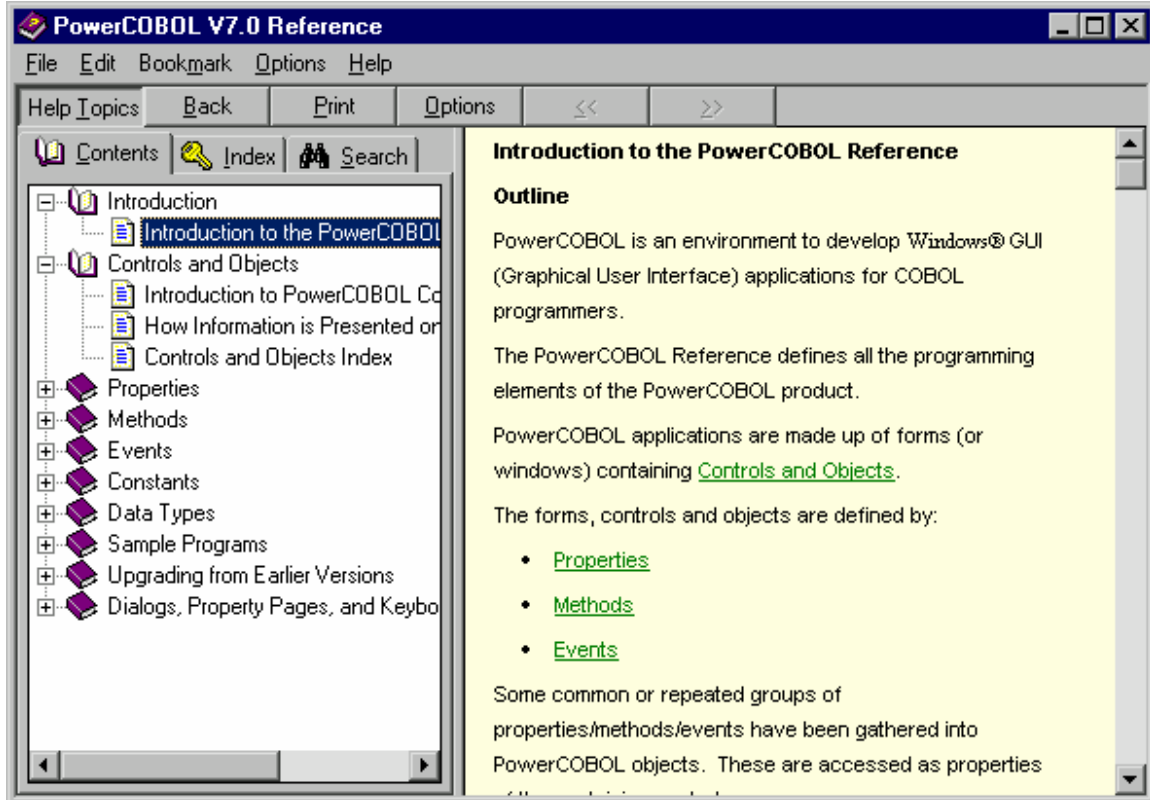


Figure 10.1. The PowerCOBOL on-line help system

Note that the above figure may appear differently if you have previously accessed the help system and changed the current tab. If so, simply click on the Contents tab. Figure 10.1 shows the help system with the Controls and Objects book open. Go ahead and open the Controls and Objects category as shown in Figure 10.1.

Select the Methods option by double-clicking the left mouse button on Methods to expand it. You will see multiple topic areas become available. Select the Methods Index option.

The following appears in the right-hand pane:

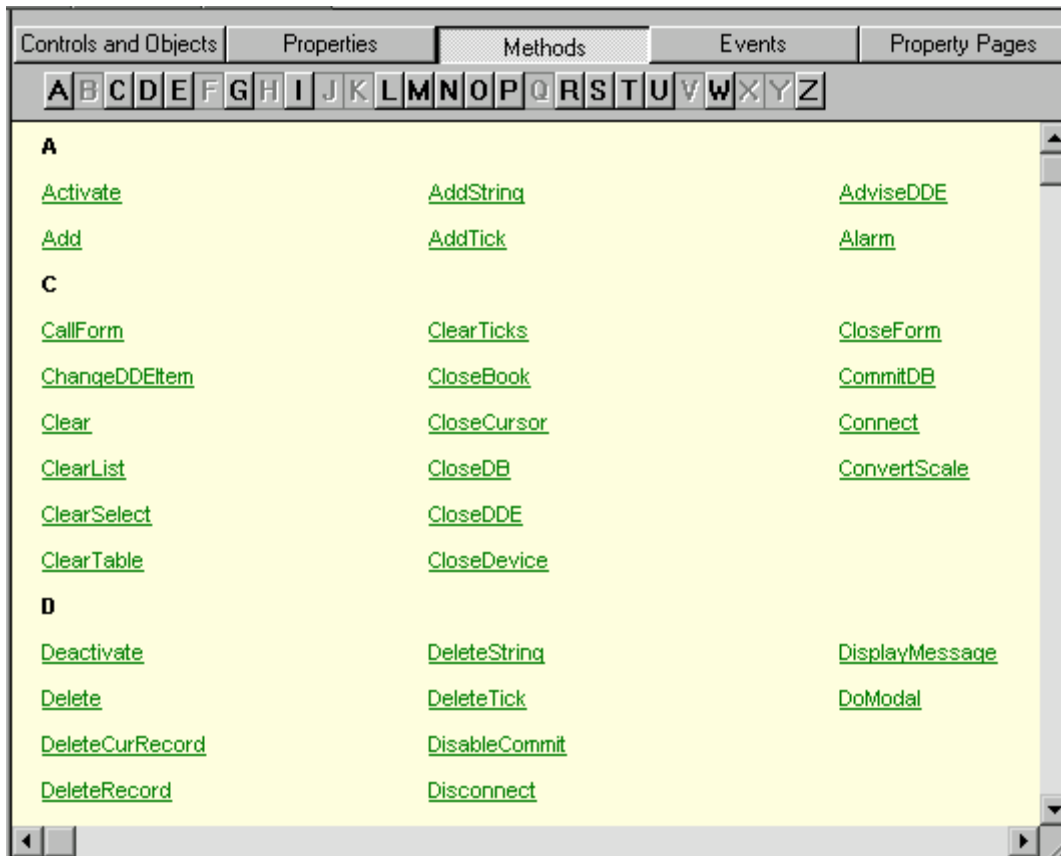


Figure 10.2. The Methods Index dialog box

This pane lists all of the current methods available in PowerCOBOL in alphabetic order. Simply click on the method you are interested in learning about to see a full explanation.

For example, if you click on the AddString method you will see:

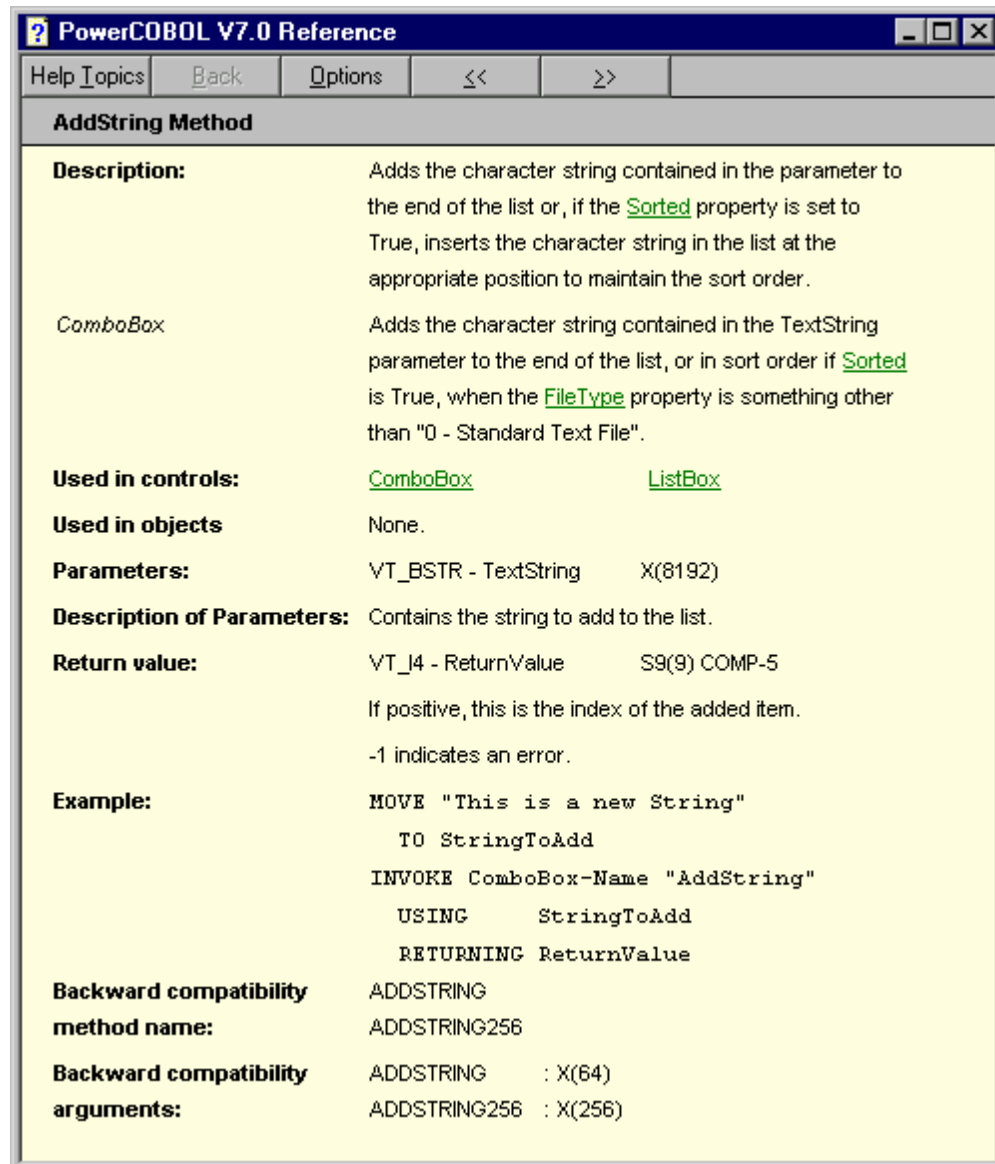


Figure 10.3. Information on the AddString Method

This dialog box gives you a description of what the method is for, a list of which controls it may be applied to, a definition of the parameters and their data types, and the possible return values. It also shows you the COBOL syntax required to use each method.

The information on backward compatibility relates to previous releases of PowerCOBOL.

In searching for information on specific methods, you may alternatively use the Find or Index tab and type in the name of the method you are searching for help on.

For example, if you are looking for information on the Clear method, you may click on the Index tab in the Help window and type in **Clear**.

As you type this in, the index will move automatically to the text item you are typing and you will be presented with:

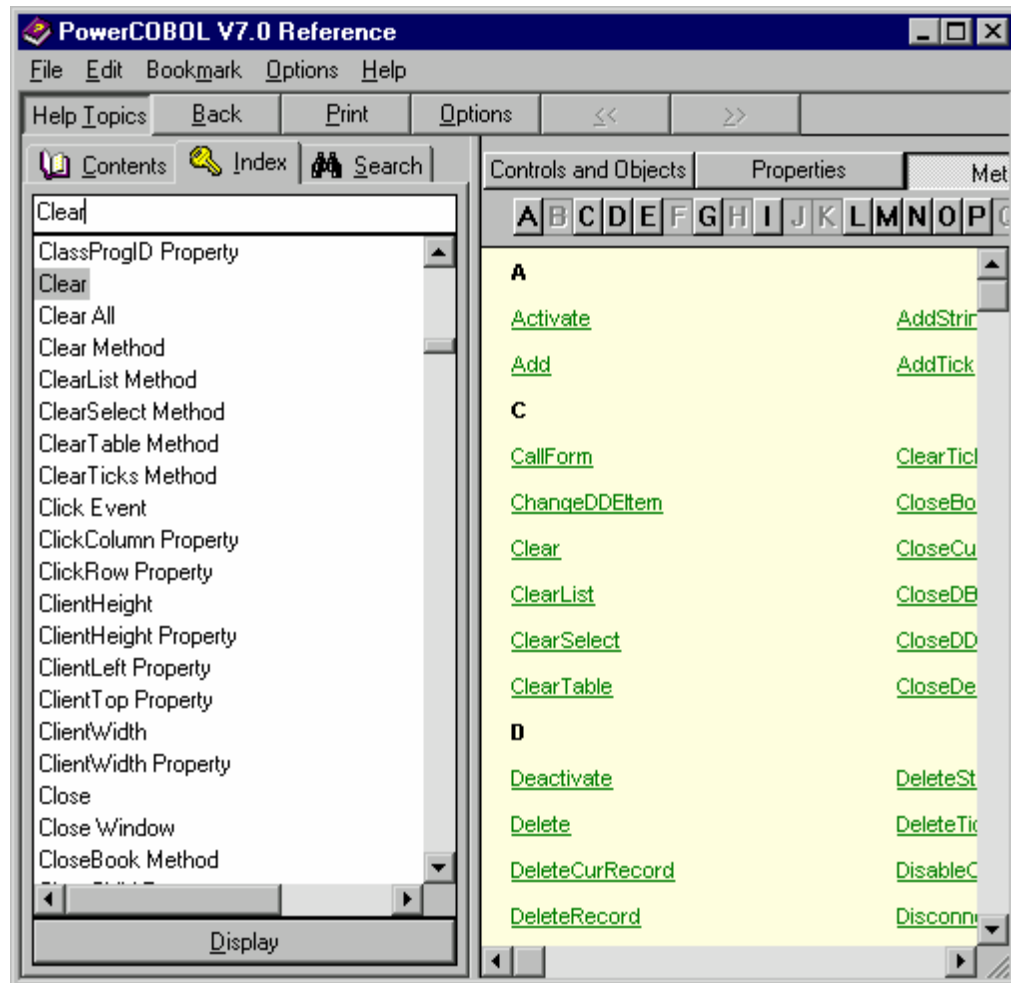


Figure 10.4. Using the Index in the help system

You may then select Clear Method in the dialog box and you will be taken to information on this method.

Using the help system, it is easy to find information on methods. You may even select, cut, and paste the code examples from the help dialog box directly into your PowerCOBOL edit sessions.

Determining Which Methods are Available for a Specific Control

The PowerCOBOL on-line help system additionally provides a useful facility for determining which methods are available for a specific type of control.

For example, if you bring up the help system and go to the Index tab and type in Static Text, you may then select the help information available for this control as follows:

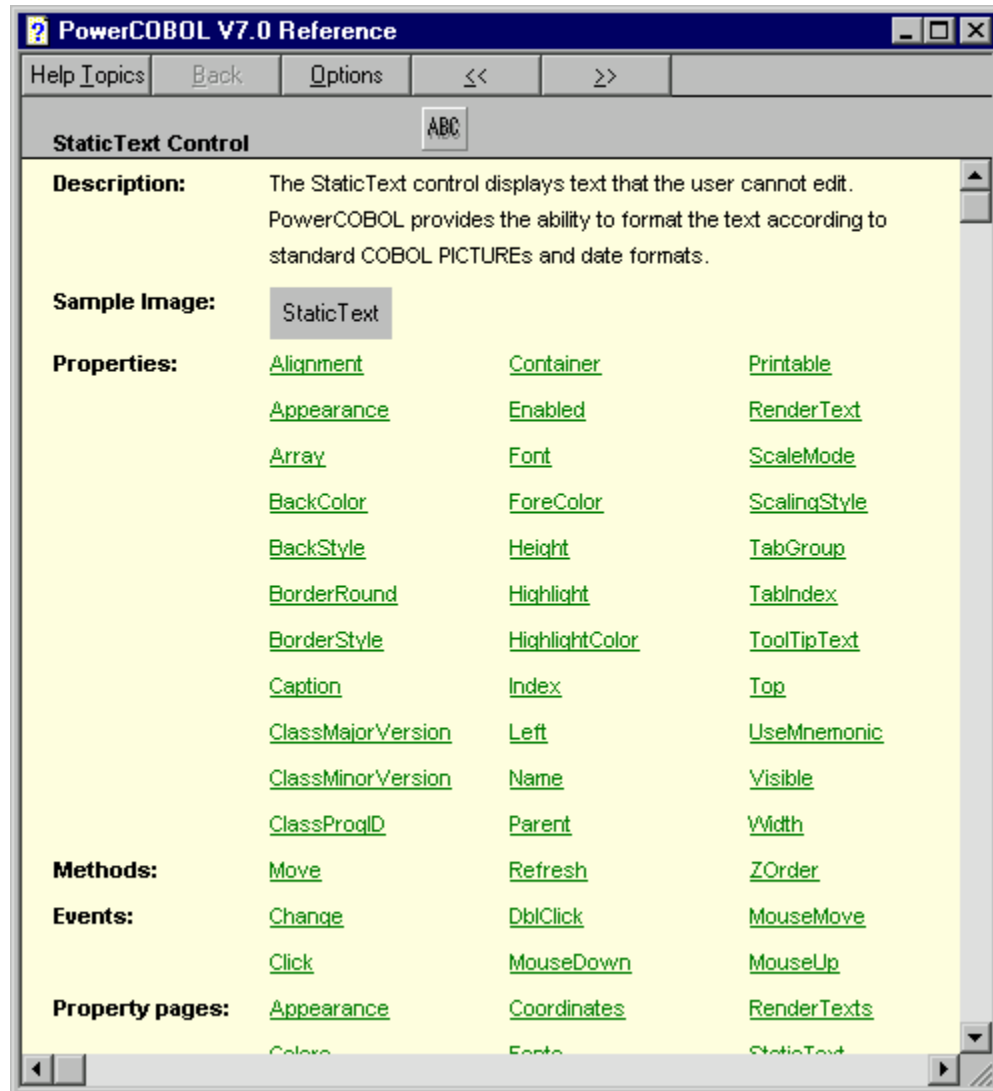


Figure 10.5. Help information for the Static Text Control

You will note in this dialog box lists of all of the properties, methods, and events available for this object.

These lists contain live linked items, and you may simply click on any one of these items to be taken directly to help information for it.

It is important to understand that methods are only available for objects (controls) when they make sense. For example, the ODBC methods are only available for an ODBC database object.

You cannot invoke an ODBC method on a static text control, as it would make no sense to do so. Therefore, the ODBC methods are not available for interaction with a static text control object.

Using Drag and Drop and Selection Lists with the PowerCOBOL Editor to Find Available Methods

The PowerCOBOL Editor provides two useful features for creating proper code to invoke methods.

Whenever you have the Form Editor active and you have an active PowerCOBOL Editor window together with it, you may use drag and drop to begin this process.

To use drag and drop, you need to be able to view the current form containing the control you are interested in. You also must be able to view at least part of the Editor window containing the location you wish to create COBOL code to invoke a method.

Move the cursor in the Editor window to the location you wish to create the COBOL INVOKE statement for the method. Next, move the mouse pointer over the Form and left-click on the control you wish to invoke a method on.

Holding the left mouse button down, drag the control (or form) into the Editor window and release the mouse button to drop it onto the edit session. Don't worry - this will not move an actual control to a new location within the form.

The name of the control you dragged into the Editor window will now appear at the current cursor location.

For example, if you had a form open in the Form Editor that contained a command button named "ButtonOK" on it, you could create an event procedure for its click event.

While the edit session is open for the click event, you could move the mouse back to the form and drag the command button (or any other control) over to the Editor window and drop it on the Editor window.

The Editor window would then appear as follows:

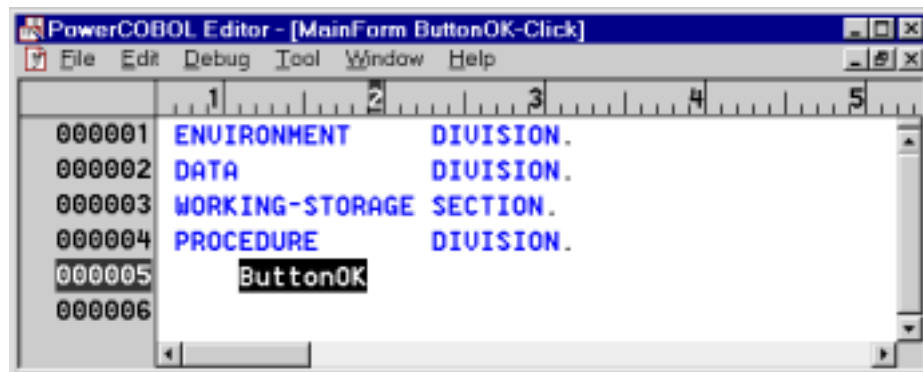


Figure 10.6. The Editor window after dragging a control and dropping it

You may alternatively simply type in the name of the control. The drag and drop feature is especially useful when you do not remember the name of the control or when it has a long name.

Once you have the name of the control in the Editor window, highlight it as shown in figure 10.6 above. Now right-click the mouse on it and select Insert Method from the pop-up menu.

For a command button such as the one shown in figure 10.6 above, you will be presented with the following pop-up menu listing all of the available methods for this control as follows:

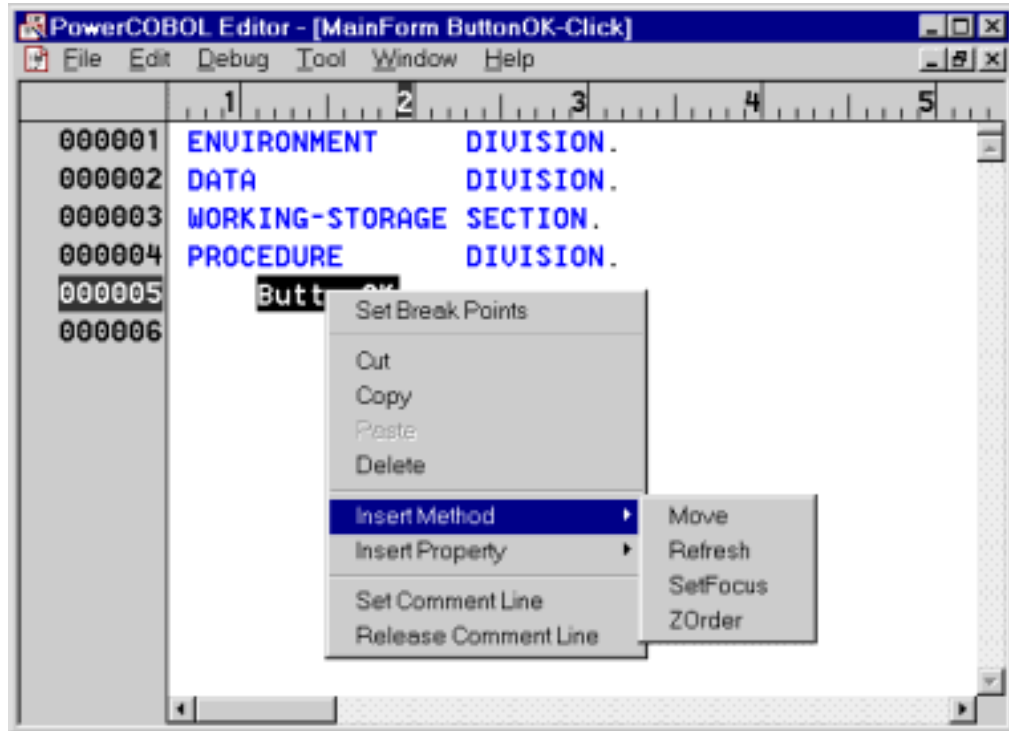


Figure 10.7. The list of methods available for a command button

It is important to note that the list of methods presented will only include methods available for this particular control.

If you select one of these methods such as "Move", a skeletal INVOKE statement will automatically be added to your procedure code such as:

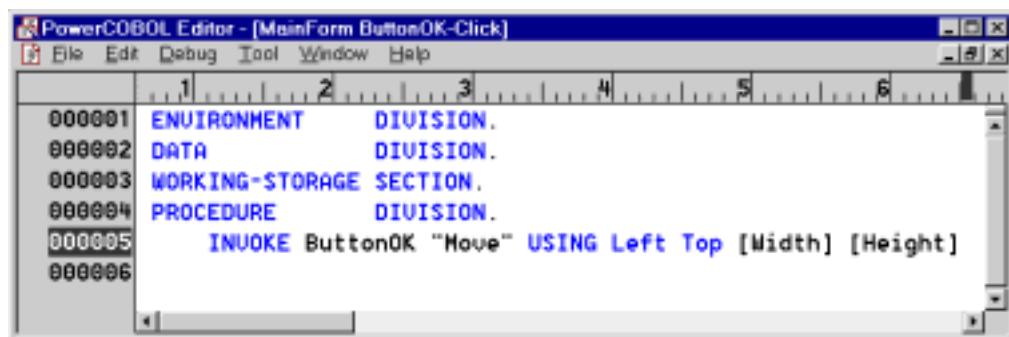


Figure 10.8. A skeletal INVOKE statement for the Move method

The skeletal INVOKE statement is in the proper format and lists both the required and optional (shown in brackets) parameters for this method.

The Move method will move the command button to a new location on the form.

You need to define parameters for the "Left" and "Top" parameters (properties) noted. The top left corner of the control will be placed at the screen coordinates you place into these parameters.

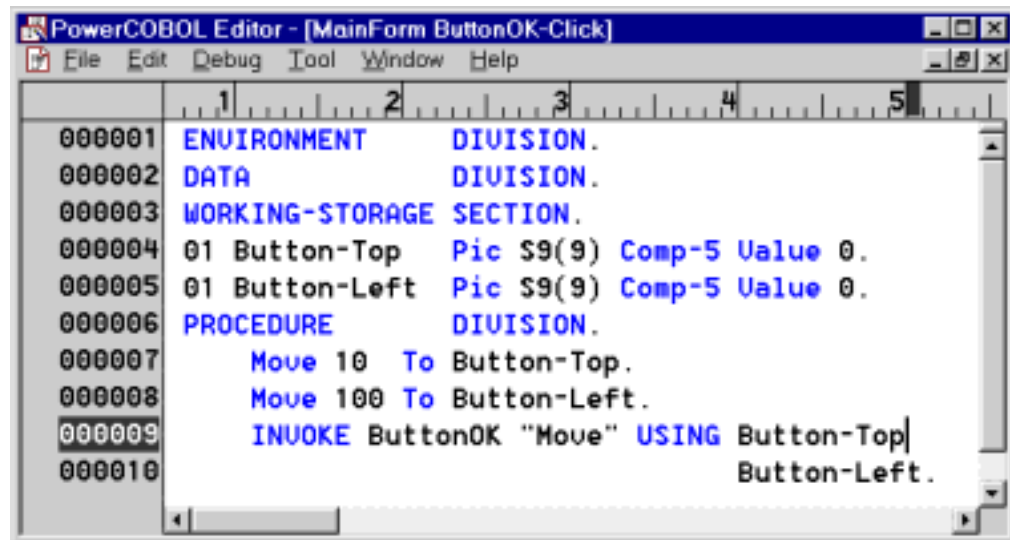
The optional "Width" and "Height" parameters will resize the control at the same time it is moved if specified.

If you do not know the appropriate COBOL data type for any of these parameters, use the Index tab in the on-line help system to look them up.

One important note about the skeletal example shown in Figure 10.8 is to realize that any word (token) that is colored in blue represents a reserved word that you may not use as a data item name.

In the example shown in Figure 10.9 above, "Left" and "Top" are both reserved words and if you attempt to create data items with the same names, you will receive compiler errors at build time.

You would thus want to modify the skeletal example to something valid such as:



```

PowerCOBOL Editor - [MainForm ButtonOK-Click]
File Edit Debug Tool Window Help
1 2 3 4 5
000001 ENVIRONMENT DIVISION.
000002 DATA DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Button-Top Pic S9(9) Comp-5 Value 0.
000005 01 Button-Left Pic S9(9) Comp-5 Value 0.
000006 PROCEDURE DIVISION.
000007     Move 10 To Button-Top.
000008     Move 100 To Button-Left.
000009     INVOKE ButtonOK "Move" USING Button-Top
000010                                     Button-Left.
  
```

Figure 10.9. A valid Move method statement

You should find the method selection lists to be useful in assisting you with choosing valid methods for controls and coding these method invocations correctly.

The same holds true for the property selection lists on the same pop-up menu.

The PowerCOBOL Sample Applications

Remember that if you need help to determine how to use a particular control or method in PowerCOBOL, have a look through the on-line help system and the included sample application projects that come with PowerCOBOL.

These samples should have been installed in the \Fujitsu NetCOBOL for Windows\COBOL\SAMPLES\PowerCOBOL folder on your system.

You should find these samples to be ready to build. You can load them into PowerCOBOL and examine them in detail.

Using the Supplied Methods

The following section documents and includes examples of how to use the supplied methods. Remember that the on-line help system is an excellent resource for looking up this information while developing applications.

Using POW-SELF as a Control

In many of the example applications supplied with PowerCOBOL, you will note that the control name "POW-SELF" appears in INVOKE statements.

You can think of POW-SELF as a name for the form you are currently editing procedure code for. POW-SELF is a convenient mechanism for not having to remember the name of specific forms when invoking methods for them.

For example, if you were currently editing one of the event procedures for a form named "MainForm", and you wanted to deactivate MainForm, you could code:

```
INVOKE MainForm "Deactivate"
```

You could also just as easily code the following to execute the identical operation:

```
INVOKE POW-SELF "Deactivate"
```

PowerCOBOL realizes the "POW-SELF" means "the form I am currently editing procedure code for", and thus "INVOKE POW-SELF" means "invoke myself".

Working with Properties

Working with Properties in the PowerCOBOL development environment is somewhat similar to working with Methods. Highlighting the name of a control in the editor such as a command button and right clicking on it will allow you to select a list of available properties for that particular control such as:

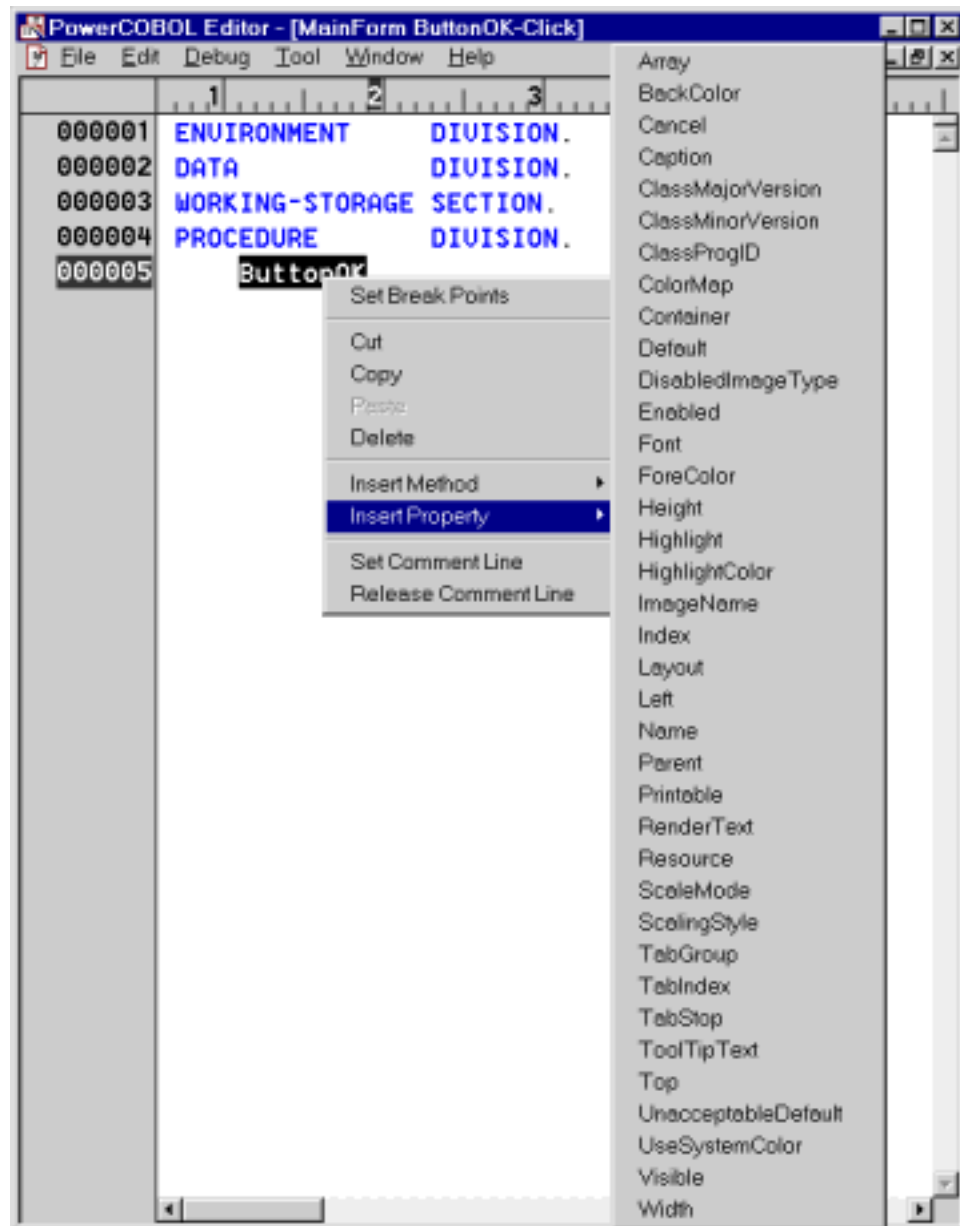


Figure 10.10 A list of available properties for a CommandButton control

As you can see, there may be many different properties available for a particular type of control. Most properties are available at both design time and runtime. For example, you could have changed a CommandButton's background color (the BackColor property) when you were painting it on a form in the form editor before you built the project. You could also change the BackColor property at runtime by moving a new color value to it such as:

```
Move POW-COLOR-BLUE To "BackColor" Of ButtonOK.
```

Note that some properties contain sub-properties and the syntax requires an additional level of direction. For example, the Font property contains sub-properties named Style, Size, Bold, Charset, Italic, Strikethrough and Underline. If you wanted

to change the size of the current font in a textbox control named CmText1 to 10 points, you might be tempted to code something like:

```
Move 10 to "Font" Of CmText1.
```

Unfortunately, the above code will generate a compile error if you attempt to build it. The reason for this error is because PowerCOBOL has no way of knowing which of the sub-properties of Font you want to direct the value 10 to. The proper code to set the font size to 10 points would instead be:

```
Move 10 To "Size" Of "Font" Of CmText1.
```

Remember to use the on-line help system to look up properties to determine if they have sub-properties and additionally to determine the valid type of data to move to them.

Some properties are available to set only at design time, such as determining if a form will have a title bar. Other properties are available only at runtime. Some properties are read only at runtime. You should become more familiar with these aspects of properties as you gain experience with PowerCOBOL.

Also note that PowerCOBOL supplies a number of system constants to use for setting properties. A color code for a background color, for example, is a bit pattern. Instead of being forced to remember and code bit patterns for colors, PowerCOBOL provides a number of constants such as POW-COLOR-BLUE that you can reference in your code. Look up Constants in the on-line help system to get a feel for what constants are available to you.

Using properties effectively can add a great deal of usability to your applications. For example, let's suppose you write an event procedure wherein if a user clicks on a CommandButton on a form, you want to print the current form. The code for this event might look like:

```
INVOKE CmPrint1 "PrintForm".
```

This code should work as expected when the user clicks on the CommandButton it is associated with. A potential problem exists however, because the nature of printers is that they are not instantaneous.

If a user clicks on the CommandButton, and the print process does not begin immediately, he or she may be inclined to click on the CommandButton again and again. This causes multiple click events and the PrintForm method will be called multiple times (once for each user click on the CommandButton).

A simple programming technique to prevent this from happening is to disable the command button after it is pushed, print the form, and then enable the CommandButton after the form has printed. Using the Enabled property of the CommandButton will do just the trick as follows:

```
Move POW-FALSE To "Enabled" Of CmCommand1.  
INVOKE CmPrint1 "PrintForm".  
Move POW-TRUE To "Enabled" Of CmCommand1.
```

You should find working with properties and methods straightforward and intuitive as you spend more time creating PowerCOBOL applications.

Chapter 11. Creating and Using Custom Controls

This chapter discusses creating and using custom OLE and ActiveX controls in PowerCOBOL applications. Custom controls are one of the most exciting and powerful additions to Windows applications.

ActiveX is a specification detailing how components should communicate with each other. It is built on top of Microsoft's COM specification, which is an object oriented design and set of low level interfaces detailing how objects developed in separate languages should communicate with each other.

ActiveX is an extension of Microsoft's OLE (some people even claim that it is a replacement for, or a newer version of OLE). For the purposes of this chapter, we will treat the terms "ActiveX" and "OLE" as one and the same.

A PowerCOBOL developed custom control typically exists as a COM enabled .DLL file. It may itself contain other controls. PowerCOBOL makes it quite simple to take almost any PowerCOBOL application and turn it into a custom control.

Controls may have events, methods and properties associated with them that are exposed to outside applications to manipulate.

Controls are in essence the basic building blocks from which many Windows applications are created. The most common uses of controls are the various graphical development elements such as command buttons, check boxes and text boxes.

Each time you place a command button on a form, you are using a control. You need not worry about creating the actual low-level code to create the graphical element and manage its low level behavior. Instead, this behavior is managed automatically.

There are many instances in application development where developers desire to construct an application using pluggable, already tested software components. This is typically what you are doing as you design a PowerCOBOL form within the form editor, by dropping controls (application components) onto a form.

Custom controls may consist of simple graphical design elements such as a command button, or more complex controls such as timers, animation controls, or a print control.

Custom controls are not required to be simple graphical development objects as well. A custom control may be an entire highly complex application that you may wish to reuse within other applications. It may also be invisible at runtime to the end user.

An example of a more sophisticated control is Microsoft's Rich Text Box control. This control is an extension of the standard text box control. Using it, you may display text using multiple colors, fonts and sizes in the same rich text box. The rich text box control also does away with the 64K size limitation of a standard text box.

A control may also be an application within itself. For example, let's suppose you wanted to create a set of SQL routines to manage data access to a database. You could create a PowerCOBOL application that performs all of the database I/O, error checking and logging.

You could then build this application as a control and register it to Windows to make it available to other applications as a control. Other Windows applications could then insert this control into themselves and utilize it.

One of the beauties of creating and using custom controls is that these controls may typically be shared between different development languages and environments within Windows. For example, you can use PowerCOBOL to create a custom ActiveX control to insert into a web page being developed using Microsoft's FrontPage.

You could just as easily insert a PowerCOBOL developed custom control into a Visual Basic application.

Note that in order to deploy a custom control which has been created using PowerCOBOL, the PowerCOBOL runtime system must be available on the client machine. Your installation process on the client machine must also register the control with Windows on the client machine to make it available for use on that machine. Windows comes with a standard utility named REGSVR32.EXE that you can call and provide the name of a COM enabled .DLL file to register to the system as a control.

In this chapter, you will taken through the process of creating a simple, yet useful custom control that you can use in any of your applications.

This control will initially be a simple clock control with a display showing a digital clock that actually ticks seconds in real time.

You will then enhance this control to include the current date then enhance it further using custom methods to provide additional functionality and interaction with an application that wishes to make use of this control.

You are free to use this control in any of your future applications if you find it useful.

Using a Form to Design a Control

While most controls are visually available (can be seen by an end user on the monitor, such as a push button), it is possible that you may create a control that has no visual property (cannot be viewed by the end user).

For example, a control that you might develop to perform file access and logging may not display any information to the end using on the monitor when the application is executing.

However, any time you develop any type of control using PowerCOBOL, you still use a PowerCOBOL form as the "container" for the control – even for invisible controls.

That is, you must still create a blank form and then create event procedures and other needed code to create a control - even if the form is never displayed.

The name you give this form typically becomes the name of the control when the .DLL containing the form is registered to the Windows system. To create multiple controls within a single .DLL file, you thus create multiple forms within the same PowerCOBOL module.

Testing Controls as they are being Developed

Because finished controls must be built as DLL files in order to be registered as ActiveX/OLE controls, you have two options for testing them as you develop them.

Because you cannot execute a DLL file directly in the Windows environment, the simplest approach is to initially develop the control as an EXE (stand alone executable application), instead of as a DLL. This allows you to execute the control as a stand-alone application and thus build it and execute it as needed to test it out. This method is desirable, but is only appropriate for controls that can behave as standalone applications.

Some controls, however, are not suitable to develop as standalone controls and you will have to write a test application that actually uses your new control to test it appropriately. However, the clock control you will build in this chapter will initially behave in this manner, so you may initially build it as an EXE application. This is because this control will initially simply display a digital clock that ticks each second, and requires no interaction from the application it is being plugged (dropped) into.

For controls that are actually designed to be interactive with another application (controls that are actually driven with method calls from another application they are intended to be dropped into), initial creation as an .EXE application will generally not be appropriate if you wish to test all of the methods out.

Instead, these controls will typically have to be built as true controls (.DLL files), and a separate driver application will have to be developed into which the control will be dropped and executed from to test its behavior.

PowerCOBOL makes handling multiple projects a snap, and this is thus a straightforward process, as will be illustrated later.

Developing the Initial Clock Control

The initial clock control that you are going to develop will be a simple numeric display of the current hours, minutes and seconds on a small form.

You will make use of a PowerCOBOL timer control, which will repeatedly generate an event at a given interval based upon how often you specify to generate the event.

The timer control is an example of a useful feature not found in traditional COBOL. In order to implement a ticking clock, your COBOL program needs to constantly check the system time and move it to the clock's display. About the only way to write such an application in traditional COBOL would be to code an infinitely running loop that accepts the current time and moves it to the clock's display.

This would be a very inefficient program and would attempt to drag down system resources by having an infinite loop in it. Instead, we need some way to put our application to sleep to release resources back to the operating system, but to have the operating system wake it back up less than each second so it can update the time.

A timer control provides a perfect solution to this need. It is one of the simplest of all controls. It contains only two properties that you will typically use. One of these properties is named Interval, which sets how often the timer will wake the application up in milliseconds. The second property is named Active and when true,

the timer is active and will wake the application up at the set Interval. When inactive, the timer is not functioning. This allows us to turn the timer on and off when needed.

Because we want the clock to run (tick) constantly, we will make the timer active from the moment the application starts. You will want to generate this event much faster than at one second intervals to ensure that the application is given control and has time to query the system time and update it in the display before the next second occurs. This means we want to set the Interval property to a value less than one full second in length.

The timer control actually wakes our application up by generating an event, and we simply code an event procedure for this timer event to update the clock. It thus becomes a very simple and straightforward application.

Creating the new Clock Project

You should now initialize the PowerCOBOL environment and create a new ActiveX project using the New option from the File menu and selecting the ActiveX template.

Go ahead and expand the project to show all of its components:

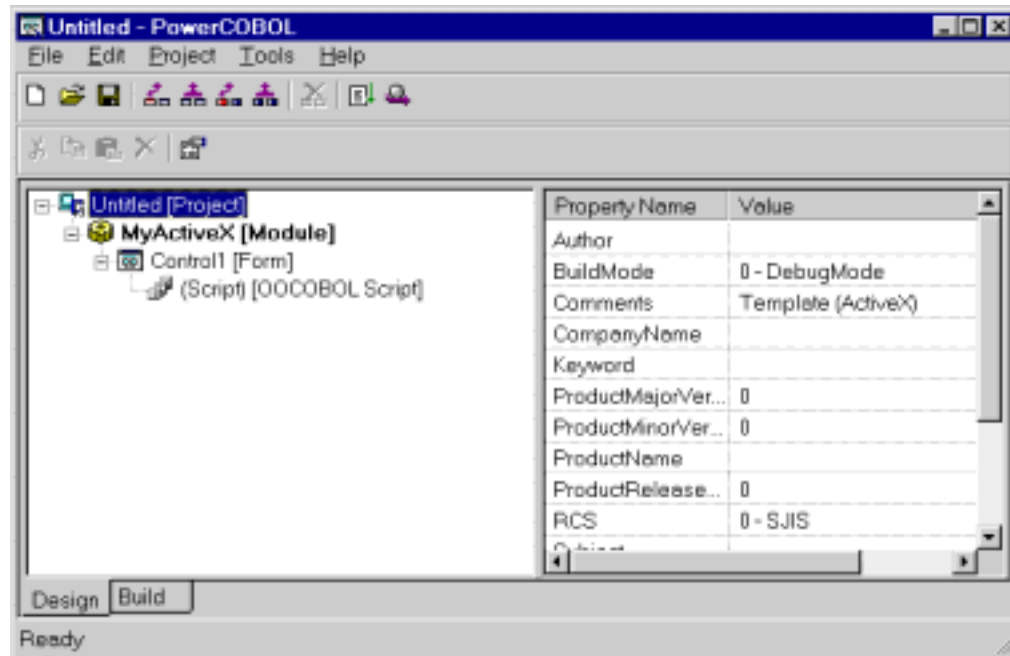


Figure 11.1. The new ActiveX project

Rename the newly created module from "MyActiveX" to "MyClock" by right clicking on it and selecting the Rename option from the pop-up menu.

Also note the gear icon to the left of the module name. This indicates that PowerCOBOL will build a .DLL file for this module. Because we want to test this as a stand-alone .EXE file first, select the module and change the FileType property to 0 - Execute Module. The gear icon should change to an application icon.

Rename the newly created form from "Control1" to "ClockForm" using the same mechanism.

Because you want to create this control so that it may be placed within another form just like any other control, you will want to do away with ClockForm's title bar. Select ClockForm and change the Titlebar property to False.

Note that you will still observe a title bar when editing ClockForm in the form editor, but this title bar will not be displayed at runtime when executing the control.

Save the project as "MyClock" by selecting Save as from the File menu and changing "Untitled.ppj" to "MyClock.ppj" in the File name field and clicking on the Save button.

The project should now appear as follows:

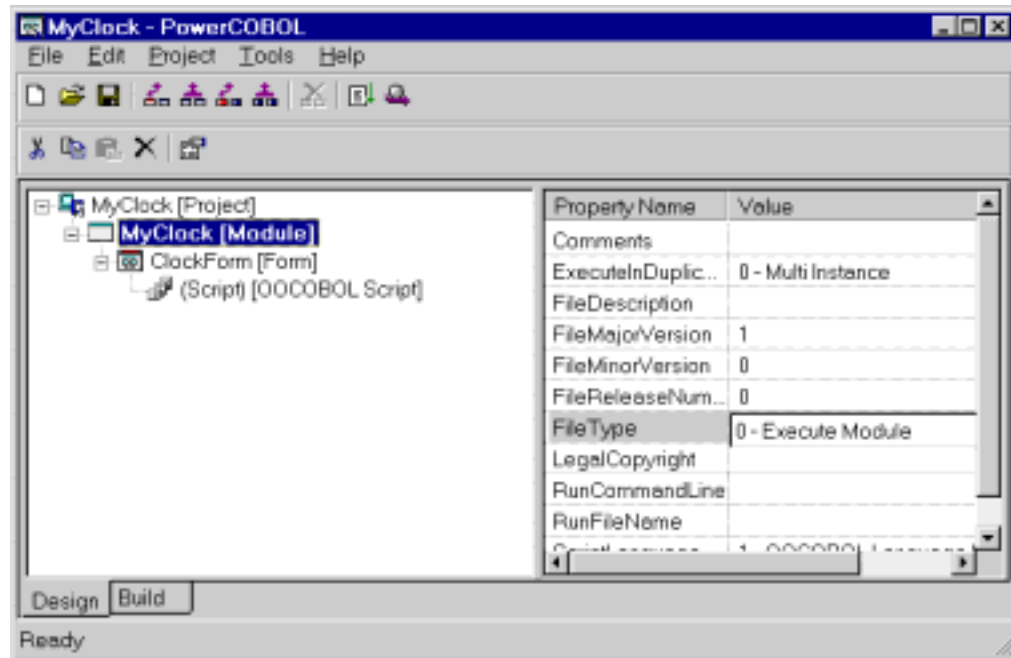


Figure 11.2. The new project with the names changed

Note that the term OOCOBOL Script appears next to the Script section in the Project Manager. When you create an ActiveX control in PowerCOBOL, the scripting language becomes Object Oriented COBOL. You may still code using the procedural COBOL that you should already be familiar with, but you can additionally use OO COBOL extensions if you know how.

OO COBOL is the default scripting language for ActiveX controls because it allows multiple instances of the same control to be active. You should leave the scripting language set to OO COBOL for ActiveX controls.

Editing the New Form

The clock display will actually be a PowerCOBOL form. This means that you need to go into the Form Editor on ClockForm, place the appropriate controls on the form and resize it appropriately.

Bring up the form editor on ClockForm by right clicking on it and selecting Open from the pop-up menu. It should appear as follows:

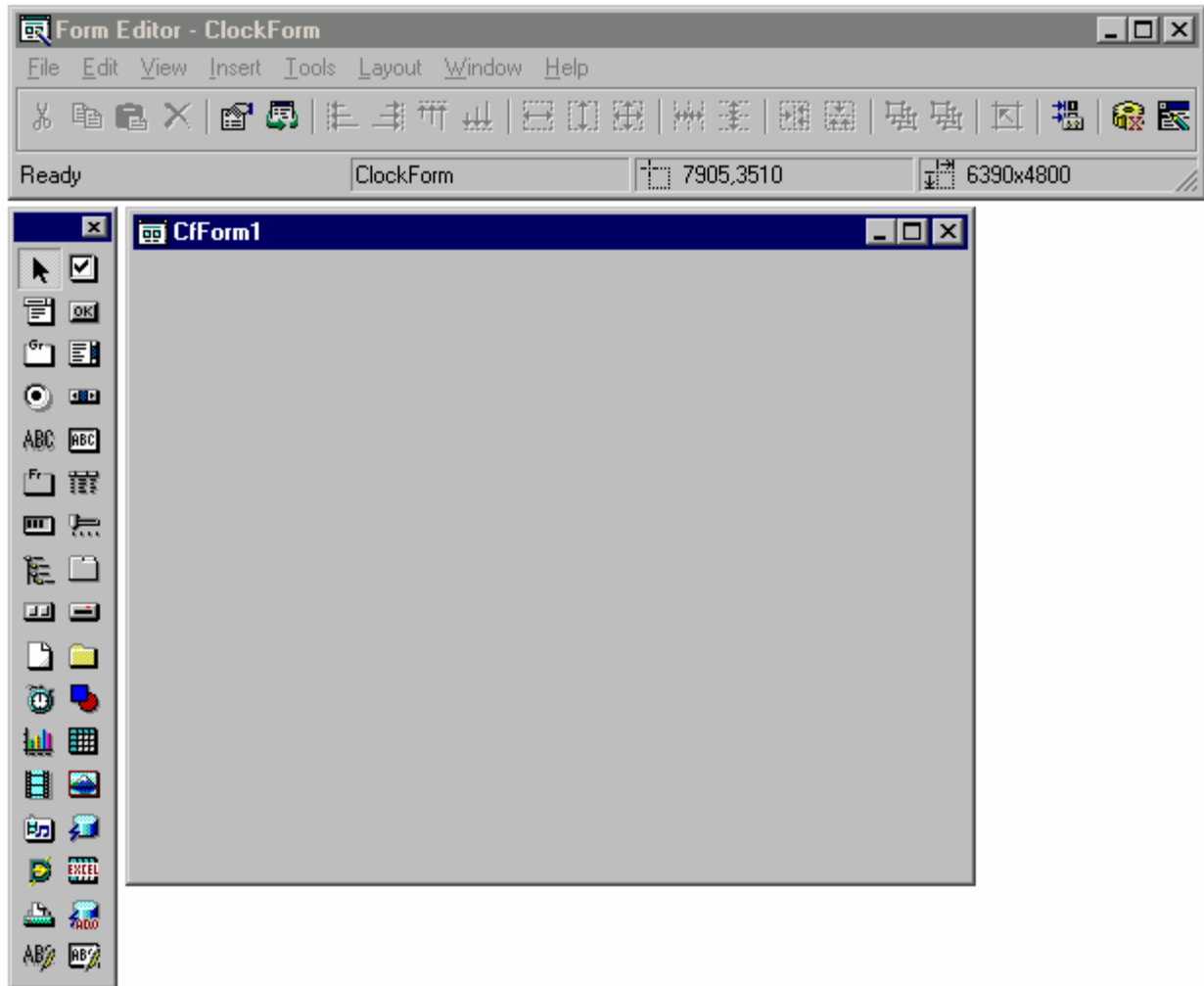




Figure 11.3. The new form in the Form Editor

Placing Controls on the New Form

You will now place a PowerCOBOL timer control and a PowerCOBOL StaticText control on the form in the upper left corner.

Once you have placed these controls on the form. You will re-size the form to fit the current time display.

Move the mouse to the Timer control  in the Toolbox palette and left-click on it. Move the mouse to the top left corner of the form and drop the timer control on the form.

Now select the StaticText control  from the Toolbox palette and move back to the form and drop a static text control in the top left corner of the form, leaving a small amount of the timer control visible so you may still select it easily.

The form should now appear as follows:

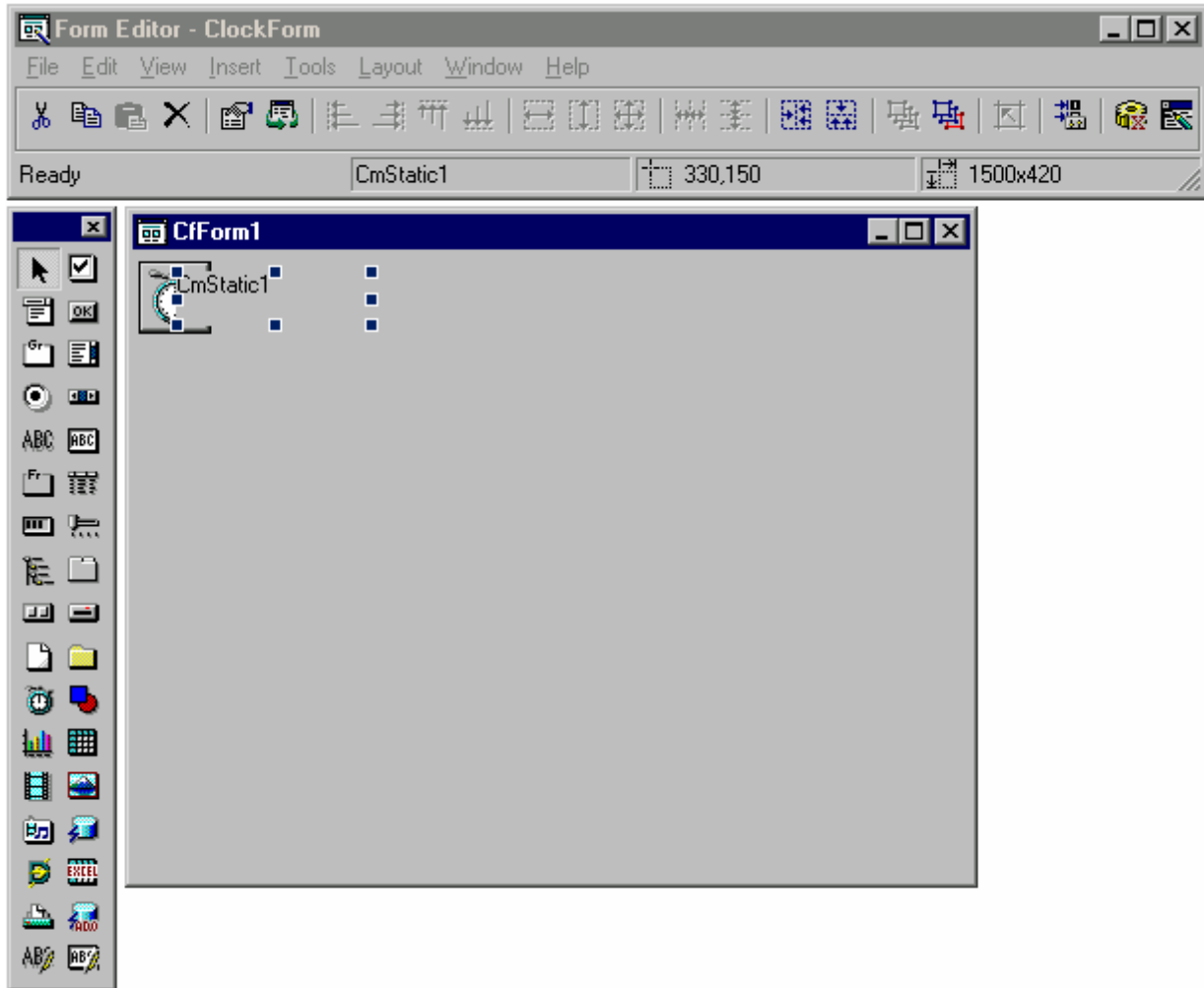


Figure 11.4. The new form with the timer and static text controls placed on it

Now move the mouse to the bottom right corner of the form and hold the left mouse button down on it. Drag the borders of the form towards the top right corner to re-size it so that it is just a bit larger than the area occupied by the time and static text controls.

The form should now appear as follows:



Figure 11.5 The new form re-sized to fit the controls

Setting the Timer and StaticText Control Properties

You now need to change some properties for the timer and static text controls. Right-click the mouse on the timer control in the form and select Properties from the pop-up menu.

In the dialog box that appears, set the Interval to 500 msec, and click the Active check to make the timer active when the form is executed as follows:

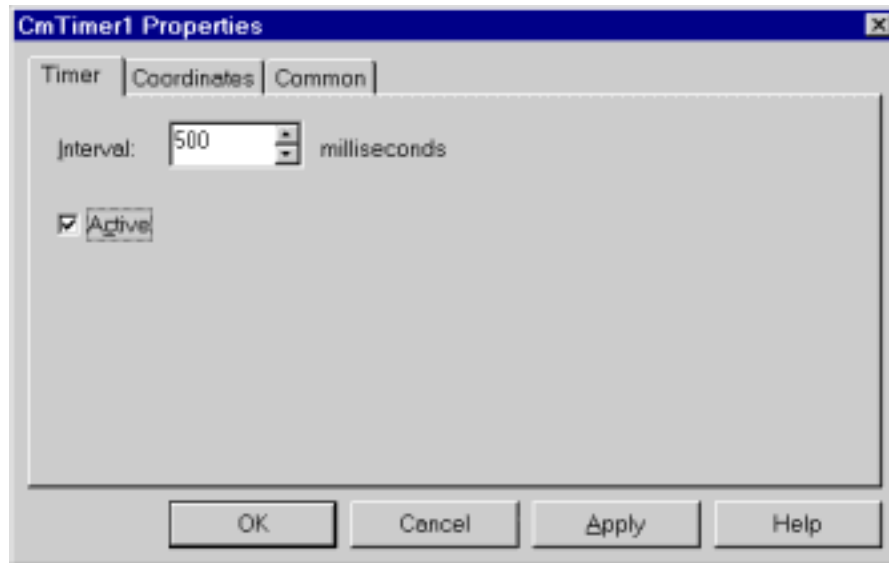


Figure 11.6. The Properties dialog box for the timer control

Click on the OK button. Now move the mouse over the static text control and right-click on it. Select Properties from the pop-up menu.

Click on the StaticText tab.

Delete the current text in the Caption property, so that this will not display when the form appears.

Click on the Horiz. Center and the Vert. Center option buttons.

Click on the Colors tab.

Under Properties, select ForeColor from the first pull-down selection list. Select Standard Colors from the second pull-down selection list and then click on Red to select red as the foreground color. This will cause the text being displayed in the static text box to be red.

Click on the Font tab. Change the size to 12, and select the Bold option under Style.

Click on the Appearance tab. Set the BackStyle property to "0 - Transparent". Set the "BorderStyle" property to "0 - Off". Set the Appearance Property to "0 - Flat".

Click on the Common tab. Change the name of this control to TimeBox. Click on the OK button to close the Properties dialog box.

You may notice that, because you turned the border off for the static text control and have blanked out its initial text value, it appears to have disappeared from the form. It is still there, however, and you can access it by simply clicking somewhere on it and an outline will appear.

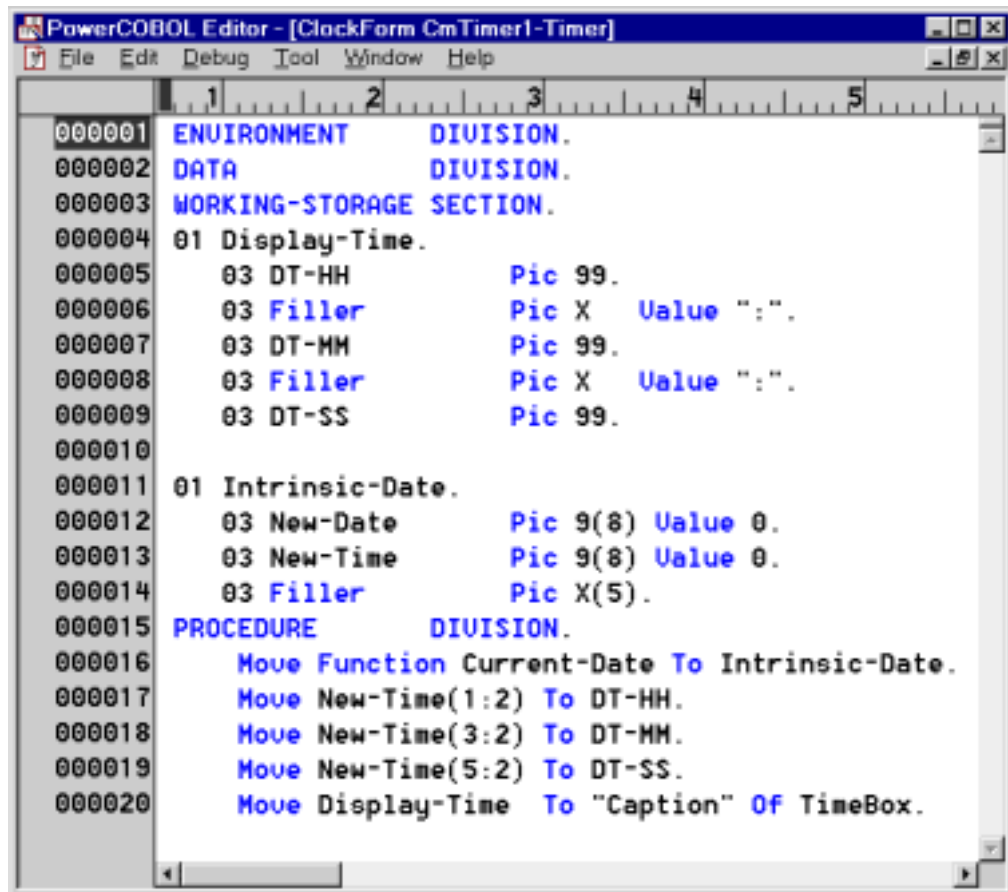
Creating the Timer Control Event Procedure

You are now ready to create the event procedure for the timer control. This is the only event procedure needed to implement the clock control.

The timer control is currently configured to generate a timer event every 500 msec. You need to create an event procedure for this event that will simply get the current time from the system and display it in the static text control. That's all there is to implementing this control.

Move the mouse over the timer control and right-click on it. Move the mouse over Edit the event procedure in the pop-up menu to bring up a second pop-up menu. Left-click on Timer.

This will bring up the event procedure for the timer event in the PowerCOBOL editor. Enter the following code:



```

000001 ENVIRONMENT      DIVISION.
000002 DATA             DIVISION.
000003 WORKING-STORAGE SECTION.
000004 01 Display-Time.
000005     03 DT-HH          Pic 99.
000006     03 Filler         Pic X   Value ":".
000007     03 DT-MM          Pic 99.
000008     03 Filler         Pic X   Value ":".
000009     03 DT-SS          Pic 99.
000010
000011 01 Intrinsic-Date.
000012     03 New-Date       Pic 9(8) Value 0.
000013     03 New-Time       Pic 9(8) Value 0.
000014     03 Filler        Pic X(5).
000015 PROCEDURE          DIVISION.
000016     Move Function Current-Date To Intrinsic-Date.
000017     Move New-Time(1:2) To DT-HH.
000018     Move New-Time(3:2) To DT-MM.
000019     Move New-Time(5:2) To DT-SS.
000020     Move Display-Time To "Caption" Of TimeBox.
  
```

Figure 11.7. The event procedure for the timer event

You will note that we have chosen to use a COBOL intrinsic function to retrieve the current time. You could have just as easily used the COBOL ACCEPT ...FROM TIME function.

We have chosen to use the intrinsic function, as it returns a great deal more of information that you will use to enhance the functionality of this control later.

Save the event procedure and exit the form editor. Go back to the PowerCOBOL project Manager for your project.

The project manager window should now appear as follows:

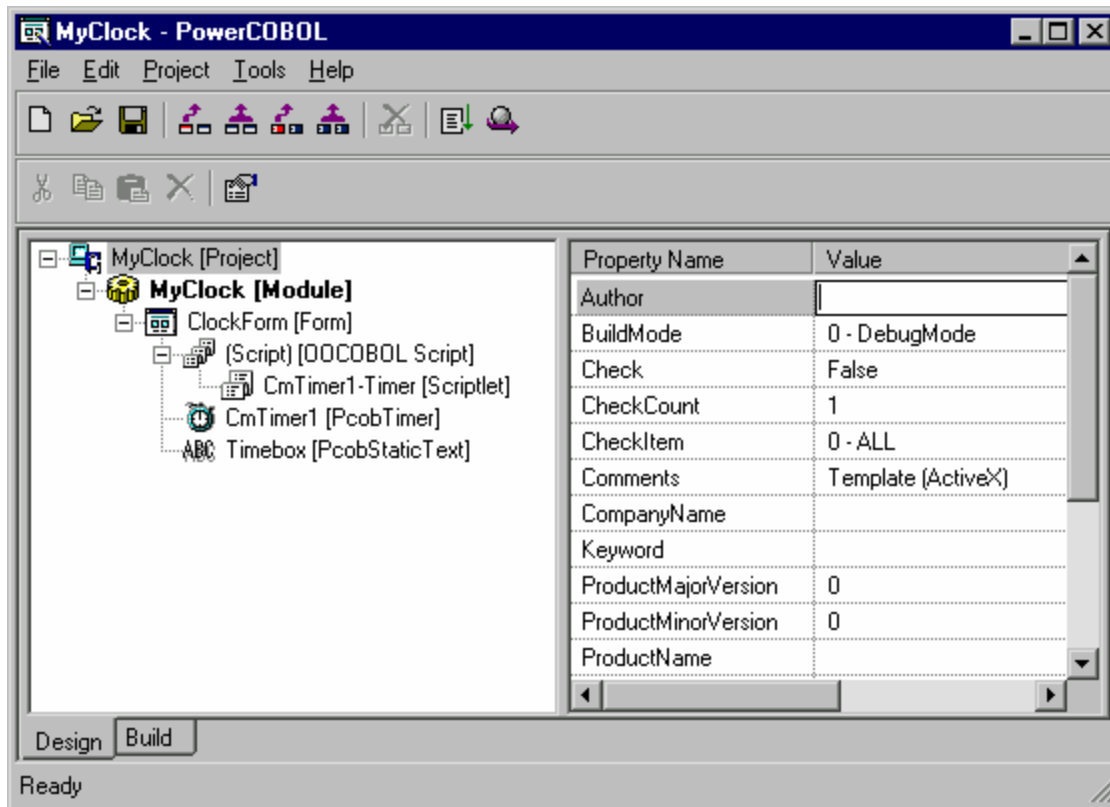


Figure 11.8. The Project Manager window after creating the event procedure for the timer event

Building and Executing the Project

You are now ready to build the first rendition of the clock control. Save the project first. Now right-click the mouse on the project name in the left window. Select Build All from the pop-up menu.

If you receive any errors during the build, go back and correct them. Remember that double-clicking on any red error messages in the build window will automatically position you in the editor on the offending line of code. Once you have a clean build, you are ready to execute the control for the first time.

Remember that you have built this as an EXE (stand alone application) this first time, so it is technically not a control (which would require it having been built as a DLL).

To execute the module, right-click on the module name in the left window and select Execute from the pop-up menu. The application should initialize and appear as follows:

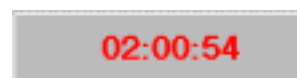


Figure 11.9. The Clock application

You may notice that there is apparently no easy way to close the clock application while it is executing. Because you have turned off the title bar, there is no close button.

Additionally, you did not place a command button or create an event to close this application.

To close the application, you need to move the mouse down to your Windows task bar, which should contain a button entitled "CfForm1". You can right-click the mouse on either of these bars and select "Close" from the pop-up menu to close the application.

Re-building and Executing the Project as a Control

In order to re-build the project as a true control, you need to change the File Type property of the module from "0 - Execute Module" to "1 - DLL Module".

Make this change now by selecting the properties for the MyClock module and changing the File Type property to "1- DLL Module".

You will notice that a new icon will appear to the right of the module name that looks like a gear. It signifies that this module is a component or control module.

Since you are going to register this as a control on your system, you should take the following steps to give it a meaningful name:

1. Select ClockForm form and right-click on it and select Properties from the context menu that appears. This will bring up ClockForm's properties.
2. Click on the OLE tab. In the ProgID property, enter:
MyClock.ClockForm

This will become the name of the control.

3. In the Description property, enter a meaningful description such as "My COBOL Clock Control".
4. Click on the OK button to close the properties dialog window.

Now save the project and rebuild it. You now have a control module. You must, however, perform one more step before being allowed to insert this control into another application.

You must *register* the control to the Windows system. You do this by right clicking on the module name (MyClock) in the Project Manager hierarchy and selecting the Register option from the pop-up menu. You should receive a message box displayed indicating that the control was registered successfully.

You cannot execute the control directly because it is a true control (you cannot execute a .DLL module in Windows). You must add this control to another project in much the same way as you add a CommandButton control to a form.

Create a new standard form project in PowerCOBOL (this will automatically close the current clock control project).

Now open the form editor on the newly created form, MainForm.

You will see the standard Toolbox palette. The clock you created exists in your Windows system as an external (custom) control that may be added into a wide variety of applications using any number of development environments.

The form editor should appear as follows:

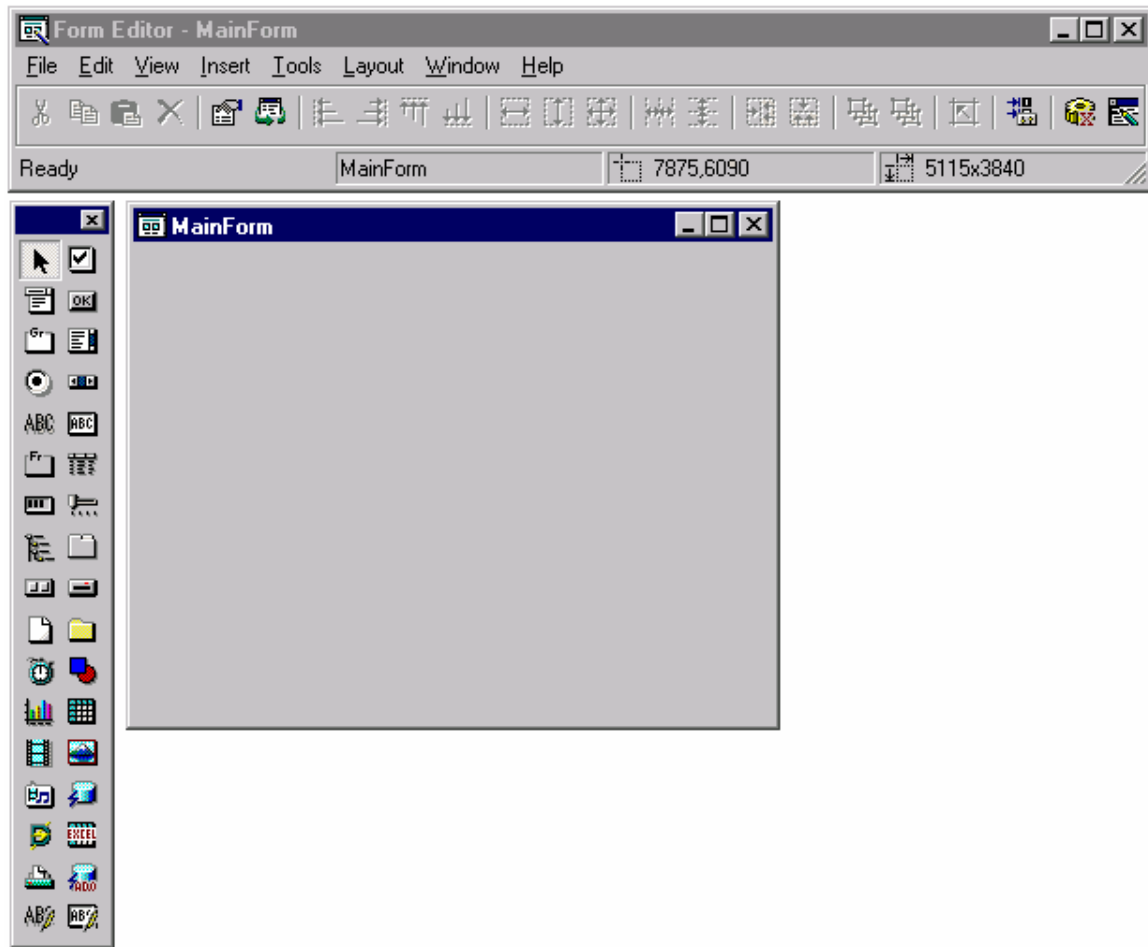


Figure 11.10. The form editor on the new form

In order to access external custom controls available in your system, you select the Custom Controls option from the Tools menu, or click on the mini icon in the tool bar that looks like a gear or right-click on the Toolbox palette and select the Custom Control option. Go ahead and right click on the Toolbox palette and select the Custom Control option from the context menu that appears.

Your system should process for a few moments (it is reading the Windows Registry and creating a list of all registered controls on your system). This will present you with a list of all registered custom controls available in your system. You may typically use any of these controls within a PowerCOBOL application.

Scroll down the list in the window presented to you and find the "MyClock" control you just built and registered, and check it to select it to add to your Toolbox palette:

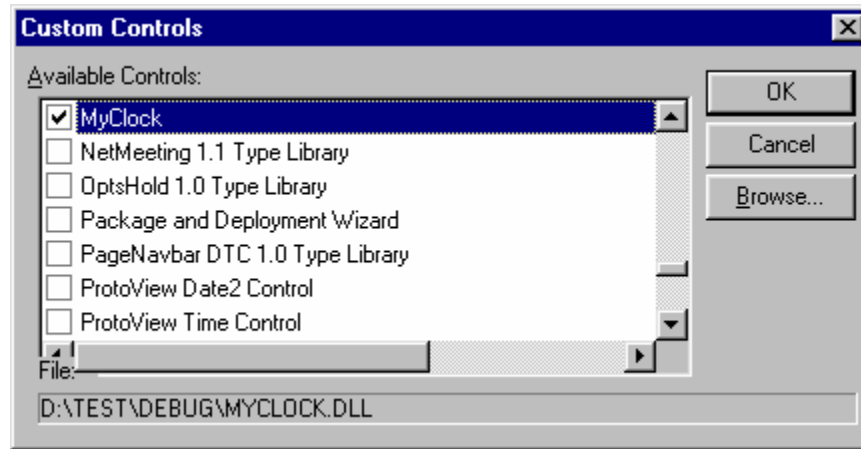


Figure 11.11. The Custom Controls dialog box

You will note that a new control appears in the Toolbox palette at the bottom entitled "OCX":



Figure 11.12. The Toolbox palette with the new clock control selected

If you would like to specify the OCX icon to be another image, use the `ToolboxBitmap` property of the form.

You may now use the clock control just like any other control by clicking on it in the toolbox palette and dropping on your form.

Go ahead and click on the OCX icon in the toolbox palette and move over to the form and click on it to drop the control on the form.

The Clock control should appear and should begin ticking on your form. You may have to resize it to see all of it.

The Form should now appear as follows:

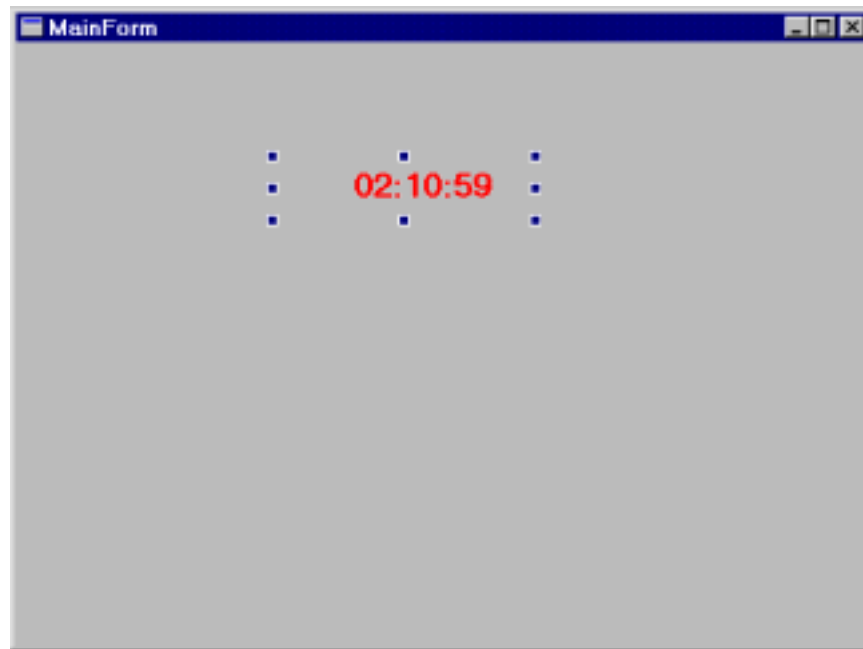


Figure 11.13. The new form with the clock control placed on it

Because the background colors of the control and the new form match, you see only the time text displayed on the new form. The StaticText control on the clock control had its background color scheme set to transparent and its border turned off, so it blends in with the form.

There are a number of properties and methods of the control that are externally accessible for the control and are thus able to be manipulated at runtime from any application in which the control has been dropped into.

In order to determine which properties and methods are available at runtime, you enter and select the name of the control into the PowerCOBOL editor (when you are editing an event procedure, for example) and right-click on the selected control. You may then select Insert Method or Insert Property from the pop-up menu. For the clock control you have just inserted into the new form, the externally available properties are:

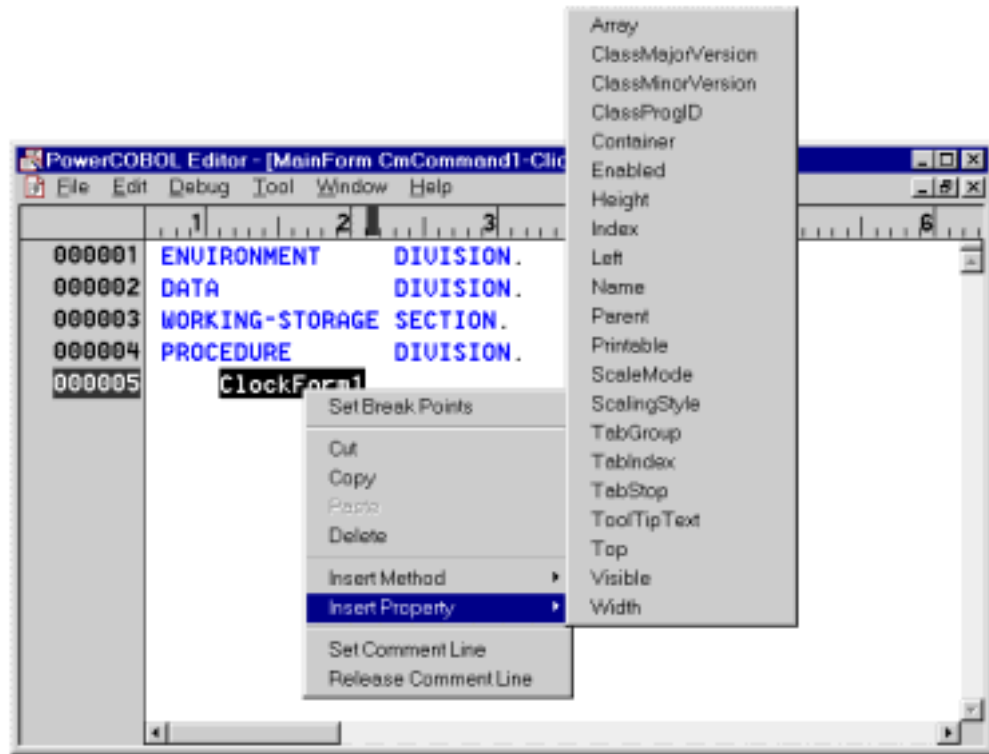


Figure 11.14. External Properties available for the clock control

You should now begin to realize just how powerful component level programming using controls truly is. You can drop controls into your PowerCOBOL projects from a wide variety of resources.

You may have noticed when viewing the custom controls list to find the MyClock control that there are a number of other controls available on your system.

You may now experiment with the new project you have created and dropped your external clock control onto.

When you are finished, save this project, naming it ClockTest and close it.

Enhancing the Clock Control

In this section you are going to enhance the clock control, by adding the current date and day of the week. You will also make use of some additional properties to enhance the appearance of the control.

Bring up PowerCOBOL and open the MyClock project containing the clock control your previously developed:

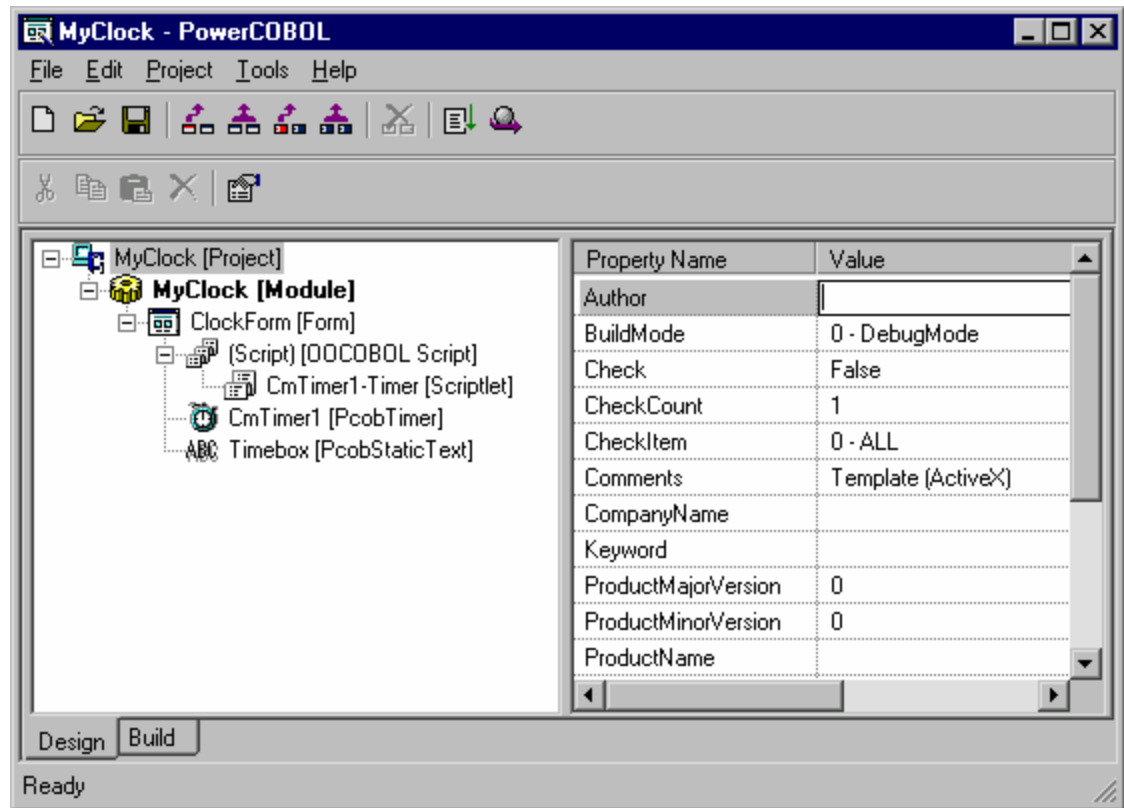


Figure 11.15. The initial clock control project

Open the ClockForm Form in the form editor. You are going to enlarge the size of the form and add an additional static text control to it. You will move the current static text control to the bottom of the form.

The new static text control you will place at the top of the form will be used to display the current day of the week and the current date. You will name this static text control "DateBox" and blank out its current caption property value to make it blank.

The bottom static text control will be the existing static text control that will continue to display the current time.

Go ahead and enlarge the form, move the current static text control down to make room for a new one above it, and add the second static text control to the top of the form as shown in figure 11.16 below.

Now move the mouse over the new static text control at the top of the form and right-click on it. Select Properties from the pop-up menu.

Click on the StaticText tab.

Delete the current text in the Caption property, so that this will not display when the form appears.

Click on the Horiz. Center and Vert. Center option buttons.

Click on the Font tab. Select the Bold option under Style.

Click on the Appearance tab. Set the BackStyle property to "0 - Transparent". Set the "BorderStyle" property to "0 - Off". Set the Appearance Property to "0 - Flat".

Click on the Common tab. Change the name of this control to DateBox. If the Replace utility dialog box appears, select OK to have it replace the name in your project.

When you are finished click on the OK button to close the Properties dialog box. The form containing the clock control should appear as follows (the outlines indicate where the static text fields are located):

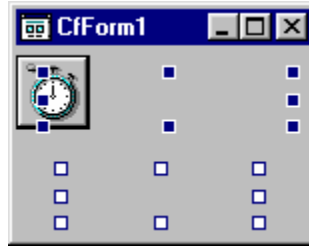


Figure 11.16. The ClockForm enlarged and with a second static text control

You are now ready to update the timer control event procedure to add the current day of week and date functionality to the control.

Updating the Timer Event Procedure

As noted above, the COBOL intrinsic function currently being used to retrieve the time retrieves other valuable information, including the current date and day of the week.

Bring up the event procedure code for the timer event in the PowerCOBOL editor. You can do this in the form editor by right clicking on the timer control and selecting Edit the event procedure and then selecting Timer.

Modify the event procedure as follows:

```

ENVIRONMENT      DIVISION.
DATA             DIVISION.
WORKING-STORAGE SECTION.
01 Display-Time.
   03 DT-HH      Pic 99.
   03 Filler     Pic X Value ":".
   03 DT-MM      Pic 99.
   03 Filler     Pic X Value ":".
   03 DT-SS      Pic 99.

01 Display-Date.
   03 DD-MM      Pic 99.
   03 Filler     Pic X Value "/".
   03 DD-DD      Pic 99.
   03 Filler     Pic X Value "/".
   03 DD-YY      Pic 9(4).

01 Intrinsic-Date.
   03 New-Date   Pic 9(8) Value Zeroes.
   03 New-Time   Pic 9(8) Value Zeroes.
   03 Filler     Pic X(5).

01 Days-Of-Week.
   03 Filler Pic X(9) Value "Monday".
   03 Filler Pic X(9) Value "Tuesday".
   03 Filler Pic X(9) Value "Wednesday".
   03 Filler Pic X(9) Value "Thursday".

```

```

03 Filler Pic X(9) Value "Friday".
03 Filler Pic X(9) Value "Saturday".
03 Filler Pic X(9) Value "Sunday".

01 Months-Of-Year.
03 Filler Pic X(9) Value "January".
03 Filler Pic X(9) Value "February".
03 Filler Pic X(9) Value "March".
03 Filler Pic X(9) Value "April".
03 Filler Pic X(9) Value "May".
03 Filler Pic X(9) Value "June".
03 Filler Pic X(9) Value "July".
03 Filler Pic X(9) Value "August".
03 Filler Pic X(9) Value "September".
03 Filler Pic X(9) Value "October".
03 Filler Pic X(9) Value "November".
03 Filler Pic X(9) Value "December".

01 Current-Day-Of-Week Pic 9.
01 Day-Offset          Pic 99.
01 Month-Offset        Pic 999.
01 Old-Date            Pic 9(8) Value Zeroes.

01 Date-String Pic X(30).

PROCEDURE DIVISION.
  Move Function Current-Date To Intrinsic-Date.
  If Old-Date Not Equal New-Date
    Move New-Date(1:4) To DD-YY
    Move New-Date(5:2) To DD-MM
    Move New-Date(7:2) To DD-DD
    Move Spaces to Date-String
    Accept Current-Day-Of-Week From Day-Of-Week
    Compute Day-Offset = (Current-Day-Of-Week * 9) - 8
    Compute Month-Offset = (DD-MM * 9) - 8
    String Days-Of-Week(Day-Offset:9) Delimited By Space
           ", " Delimited By Size
           Months-Of-Year(Month-Offset:9) Delimited By Space
           " " Delimited By Size
           DD-DD Delimited By Size
           ", " Delimited By Size
           DD-YY Delimited By Size
    Into Date-String
    Move Date-String to "Caption" of DateBox
    Move New-Date To Old-Date
  End-If.
  Move New-Time(1:2) To DT-HH
  Move New-Time(3:2) To DT-MM.
  Move New-Time(5:2) To DT-SS.
  Move Display-Time To "Caption" OF TimeBox.

```

Save the event procedure code you've just entered and exit the form editor. In order to quickly test the control, change the MyClock module's File Type property to "0 - Execute Module".

Save the project and re-build it. Execute the MyClock module.

The control should now appear as follows with the exception of the actual date and time from your system:

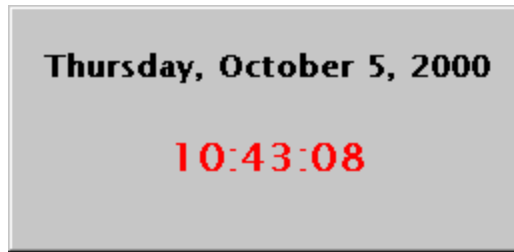


Figure 11.17. The clock control after enhancements.

You may now exit the control, go back to the project manager and change the File Type property for the module back to "1 - DLL Module". Rebuild the project.

You now have a fully functional date and clock control that may be plugged into to any ActiveX enabled application.

Using Custom Methods and Properties in Controls

You have a number of properties and methods for this control that will be directly accessible to any ActiveX enabled application it may be plugged into.

The available methods and properties, however, are only those allowed for the form containing the control itself. You cannot access properties for individual controls within a form that may contain numerous other controls.

For example, you cannot change the font size of the DateBox or TimeBox static text controls contained in the clock control from an application that is using this control, nor may you change the background color of the form containing the data and time fields.

You may, however, create custom methods and/or properties for this control that will be directly accessible to any application that wishes to include this control.

Custom methods may in turn access properties for individual controls contained within that control.

A good example usage of a custom property might be to create a custom property called "DateFormat" that specifies which format the date should be displayed in. The application using the control could then set the DateFormat property at any time, and you could add additional code in the MyClock control's timer event procedure to format the date being displayed in various ways depending on the current value of the DateFormat property.

A good example usage of a custom method, might be a method that changes the current font in one of the static text controls contained within the clock control.

This would allow an application using the clock control to manipulate its static text control font properties indirectly.

Custom Events, Methods and Properties may be inserted into a control by right-clicking on a form name in the Project Manager and selecting the appropriate option from the context menu that appears. You may also insert custom methods or properties in a form by selecting the appropriate action under the Edit menu in the Project Manager, and then selecting the Object option, or from the Insert menu of the Form Editor.

Form Level OLE/ActiveX Properties

You should also be aware that there are some form level properties available for custom controls that are specific to OLE/ActiveX (remember that we are using these terms as one and the same).

If you right click on a control's module name in the project manager and then select Properties from the pop-up menu, you will be presented with the Properties dialog box for the form.

If you then click on the OLE tab, you will see:

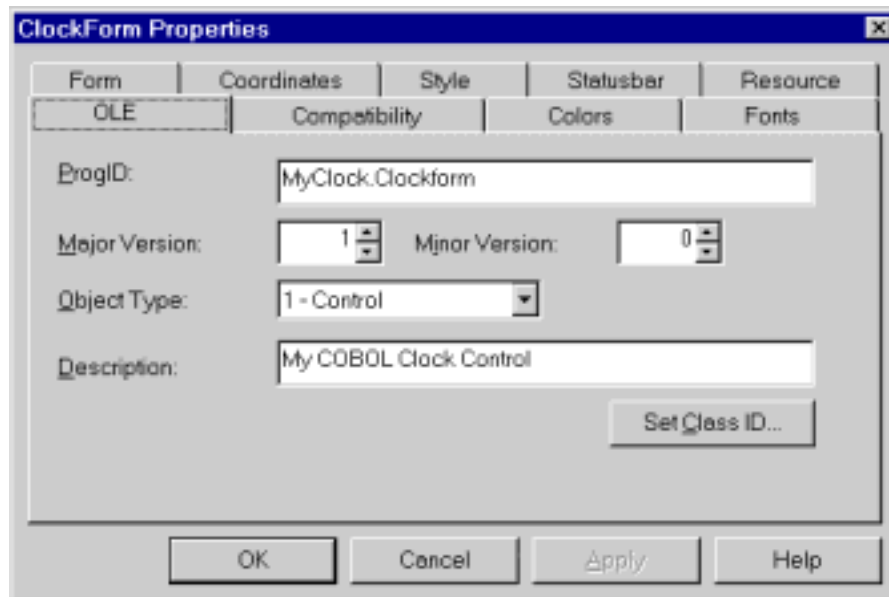


Figure 11.18. The OLE properties tab for a control module

This dialog box allows you to specify additional OLE/ActiveX properties for a specific form (control), such as a version number.

Class ID's are typically set automatically by the system and you should not modify them unless you are quite expert in what's going on at a very low level.

Using Controls to Allow Access to Multiple Forms Within a Single Module

You may use controls to obtain behavior that is not available in PowerCOBOL otherwise.

For example, you may have noticed that if you attempt to create a project that has an execution module containing 2 or more separate forms, you cannot access one form's methods and properties from another form - even though they are controlled by the same application module.

You can get around this restriction, however, if you create the other forms in a separate control and then insert them into your project module's main form.

Using Other 3rd Party Controls Within PowerCOBOL

Please note that there are literally thousands of ActiveX controls available on the market from a wide variety of vendors.

You may use these controls in PowerCOBOL applications using the same approach as noted above.

That is, you can select these controls to add to your Toolbox palette by selecting the Custom Controls option from the Tools menu or clicking on the associated mini icon in the Form Editor.

The syntax for accessing custom control methods and properties that were not created in PowerCOBOL is the same syntax as you use for accessing controls created using PowerCOBOL.

One of the most confusing aspects of using this myriad of custom controls is obtaining documentation for a certain control you may be interested in that defines available methods, properties and data types for accessing it properly.

Some controls ship with simple readme files that explain how to use them. Other controls may be shipped with a Windows .hlp (file).

If you see a file named with the extension ".hlp", simply double click in it in the Windows Explorer and it will be brought up in the Windows help system.

If you build custom controls for external distribution yourself, it's always a good idea to provide some form of on-line documentation detailing how to make use of it.

If you cannot find documentation on a particular 3rd party control, try accessing the vendor's web site if such exists.

You should also be aware that while some 3rd party controls may be freely distributed, others require a license. Check with the control's vendor or documentation to determine this.

OLE/ActiveX Data Types

OLE/ActiveX defines a standard set of data types for all controls to make use of. The naming format in manuals and other forms of documentation is typically "VT_xxx".

The available OLE/ActiveX data types and their COBOL equivalents are as follows:

VT_I2	- S9(4) COMP-5
VT_I4	- S9(9) COMP-5
OLE_COLOR	- S9(9) COMP-5
VT_BOOL	- S9(4) COMP-5
VT_BSTR	- X(8192) - maximum length
VT_CY	- S9(14)V9(4)
VT_R4	- COMP-1
VT_R8	- COMP-2
VT_DISPATCH	- OBJECT REFERENCE

OLE_???_CONTAINER	- COMP-1
OLE_???_PIXELS	- S9(9) COMP-5
OLE_???_HIMETRIC	- S9(9) COMP-5

Note: VT_I2 can be set from -32768 to 32767, but VT_BOOL can only be set to 0 or -1.

??? Indicates XPOS, YPOS, XSIZE or YSIZE.

Using ActiveX in HTML

An ActiveX control can be used in an HTML document being browsed by Internet Explorer. The ActiveX control needs to be created using the "ActiveX" template. See the description of "New Project dialog box" and "Creating Custom Controls" for details.

How to use the ActiveX control in an HTML Form

You need to use the "OBJECT" tag. For example:

```
<OBJECT WIDTH="120" HEIGHT="150"  
CLASSID="CLSID:F6E48B22-03A0-11D3-88F9-00000E98DD12">  
</OBJECT>
```

Specify the size of the ActiveX control to WIDTH and HEIGHT, and set CLASSID to the value received from the following operation:

1. Open the "OLE" tab in the Form's properties dialog.
2. Click the "Set Class ID..." button in the dialog.
3. Click the "OK" button in the message box, with ignoring the caution.
4. You can see the CLASSID in the "Object CLASSID".

Example

If the value of "Object CLASSID" indicates {409A0921-0793-11D3-88F9-00000E98DD12}, you code the HTML as follows:

```
<OBJECT WIDTH="120" HEIGHT="150"  
CLASSID="CLSID:409A0921-0793-11D3-88F9-00000E98DD12">  
</OBJECT>
```

"{" and "}" are not required. Refer to documentation regarding W3C or HTML for details of "OBJECT" tag.

NOTES

- If you use ActiveX controls in an HTML document, test the application completely.
- You cannot debug ActiveX controls in Windows95/98, even if you build them in debug mode.
- If you run on the WindowsNT, specify the "Executable File" to be "Iexplore.exe" and set "CommandLine" to the name of the HTML document file name Including it's full-path in the "Run" tab of the module properties.

- If the application abends or an error occurs, PowerCOBOL forces InternetExplorer to end the application. In this case, shut down the system using the Windows Task Manager, which is shown by "Ctrl+Alt+Delete" key, after ending all other applications. The reason for this is that the application modules may still be hooked by the system (e.g. DLL's may not be released by the operating system, and you will be unable to run the application subsequently or to re-link a newer version of the .DLL(s)).
- You should not use a COBOL "DISPLAY" sentence in the event procedure of an ActiveX control on Windows98.
- The MultipleInstance property of a form specifies whether to allow execution of multiple instances of a form (whether two or more instances of an application that include the form are allowed to exist in a Windows system at the same time).

The OLE Control

The OLE control allows you to execute OLE methods. It applies to controls in the COM class.

This is a special control that allows you to invoke the full power of OLE. The example discussed below concentrates on the OLE "Create-Object" method to create a PowerCOBOL Form at Run-time.

In order to use this, you must create two separate projects. The first project creates a .DLL file containing one or more forms. You must build this project and register the .DLL file to make the forms available to the second project at run-time.

In the second project, you must add the following line of code to each form's REPOSITORY section of the ENVIRONMENT DIVISION:

```
Class COM AS "*COM".
```

In the second project, every form defined needs the following data items declared in the form's WORKING-STORAGE section:

```
01 COM-FORM          OBJECT REFERENCE.
01 COM-FORM-NAME     PIC X(128) .
```

In the first project, where you create a .DLL with one or more forms within it, you need to look at the properties of each form and then look at the OLE tab. The Program-ID field will contain the OLE name you will use in the second project to refer to the form. Remember the OLE name. You may change it to something else if you like. Note that the actual OLE name you will need to remember will be "programid.1". (.1 indicates the version of OLE control.)

For example, if you create an OLE program id name of "Myprog", the actual OLE name you will reference the form by will be Myprog.1.

This is all you need to do to the first project. You can create forms, place controls on them and edit them in the normal way. When you are finished, build the project as a .DLL and register it.

When you are ready to begin creating the second project that will use the first project's form(s), remember to place the above noted declarations in the REPOSITORY and WORKING-STORAGE sections of your main form.

To invoke one of the first project's forms dynamically from the second project, you code (where "form1.1" is the name of the form in the first project you wish to invoke):

```
MOVE "form1.1" To COM-FORM-NAME
INVOKE COM "CREATE-OBJECT"
      USING COM-FORM-NAME
      RETURNING COM-FORM
INVOKE COM-FORM "DoModal"
```

Syntax

```
INVOKE COM-FORM "DoModal"
```

Example

PowerCOBOL comes with example applications in the Form subdirectory named "subform.ppj" and "mainform.ppj", which illustrate this behavior.

This concludes this chapter on building custom controls using PowerCOBOL. Feel free to experiment with what you've learned.

Index

- #INCLUDE statement, 135
- ADD statement, 141, 145
- adding printer support, 77
- application programming interface (API), 195
- array, 102
 - adding items, 103
 - copying items, 103
 - creating, 102
 - deleting items, 103
 - releasing items, 103
- BASED statement, 131
- batch editing, 96
- breakpoint, 173
- Build command, 160
- Choose Event dialog box, 122
- CLOSESHEET method, 249
- CONSTANT statement, 132
- Copy command, 94
- creating a new project, 200
- creating a project, 160
- creating an item, 90
- Data Division, 129
- data trace function, 174
- developing a simple application, 49
- DISPLAY statement, 143
- Dynamic Data Exchange (DDE), 21
- Dynamic Link Library (DLL)
 - calling, 231
- dynamic link library file, 231
- Edit menu, 24
- Edit Procedure window, 122
- editing commands, 95
- Environment Division, 129
- EVALUATE statement, 144
- event, 121
- event procedure, 121
 - writing, 129
- event procedures, 133
 - writing, 58
- event-driven programming, 19, 58, 121, 227
- events, 18
- File menu, 23
- FILE statement, 131
- FILE-CONTROL statement, 131
- Font Box
 - changing the font size, 107
 - changing the font style, 107
 - selecting a font, 106
- global variables, 213
- grouping items, 97
- Identification Division, 129
- IF statement, 143
- include files, 137
- infinite loop, 136
- insert mode, 127
- inserting an item name, 128
- keys, 126
- Make command, 160
- manipulating items, 94
- Microsoft Access, 195
 - sample database (Test.mdb), 196
- mouse, 126
- MOVE statement, 140, 145
- multimedia applications, 22
- Object Linking and Embedding (OLE), 21
- ODBC, 195

- 32 bit support, 195
- defining an ODBC source, 197
- developing a sample application, 195
- OpenBook Method, 287
- overwrite mode, 127
- Paste command, 96
- PowerCOBOL, 10
 - application components, 226
 - development environment, 20
 - event-driven programming, 227
 - programming paradigm, 224
- Procedure Division, 129
- PROCEDURE statement, 133
- programs
 - using COBOL and PowerCOBOL procedures, 249

- PROGRAM-STATUS, 136
- project
 - creating, 200
- projects, 159
- setting fonts, 106
- setting item order, 99
- setting the item style, 93
- sheet
 - opening in PowerCOBOL from COBOL 85, 242
- sheet procedures, 129
- SPECIAL-NAMES statement, 130
- Standard Query Language (SQL), 22
- STOP RUN statement, 136
- SUBTRACT statement, 142, 146
- WORKING statement, 132