

# **VISUAL PL/B PROGRAMMING**

**Sunbelt Computer Software**

This material is copyright Sunbelt Computer Software. All rights reserved. This publication may not be reproduced in any form or part by any means without prior written consent.

## **DISCLAIMER**

While we take great care to ensure the accuracy and quality of these materials, all material is provided without warranty whatsoever, including, but not limited to, the implied warranties of the merchantability or fitness for a particular use.

The sole purpose of this document is to serve as a workbook for live instruction. It is not intended as a reference manual. The student should refer to the appropriate vendor manual for reference information.

Due to the nature of this material, this document refers to numerous hardware and software products by their trade names. References to other companies and their products are for informational purposes only and all other trademarks are the property of their respective companies. It is not our intent to use any of these names generically.

Document Revision: 10.6

Copyright © Sunbelt Computer Software 2004-2023

# **TABLE OF CONTENTS**

## **Chapter One – Introduction to Visual PL/B**

Chapter Overview and Objectives .....	1-1
What is Visual PL/B? .....	1-2
Visual PL/B Application Types .....	1-4
Visual PL/B Terminology .....	1-6

## **Chapter Two – Exploring the Visual PL/B Environment**

Chapter Overview and Objectives .....	2-1
The Visual PL/B Design Environment .....	2-2

## **Chapter Three – Creating a Visual PL/B Application**

Chapter Overview and Objectives .....	3-1
Visual PL/B Design Steps .....	3-2
Step 1 – Design the User Interface .....	3-3
Step 2 – Create the Project .....	3-5
Exercise – Create a Project .....	3-8
Step 3 – Create the Source Program .....	3-9
Step 4 – Write the Source Program .....	3-11
Step 5 – Save Your Source Program .....	3-12
Step 6 – Create the Form .....	3-13
Step 7 – Drawing Controls .....	3-14
Exercise – Draw the User Interface .....	3-21
Step 8 – Setting Properties .....	3-22
Exercise – Setting Properties .....	3-30
Step 9 – Writing Code for Your Application .....	3-32

Exercise – Writing Code .....	3-37
Step 10 – Saving Your Form .....	3-38
Exercise – Saving Date/Time.....	3-39
Step 11 – Building Your Application .....	3-40
Step 12 – Running Your Application .....	3-41
Exercise – Building and Running Date/Time .....	3-42

## Chapter Four – Forms, Controls, and Menus

Chapter Overview and Objectives .....	4-1
Visual PL/B Forms .....	4-2
Setting Properties .....	4-6
Standard Controls .....	4-7
Command Buttons .....	4-12
StatText and LabelText.....	4-14
Exercise – Setting Properties .....	4-17
CheckBoxes, Radio Buttons, and GroupBoxes .....	4-17
EditText, EditNumber, EditDateTime .....	4-19
Icons and Picts .....	4-22
Exercise – Office Equipment Information.....	4-24
DataLists and ComboBoxes.....	4-29
Exercise – DataLists.....	4-33
Shape and Line Tools.....	4-34
HScrollBar and VScrollBar .....	4-36
Timer .....	4-37
Exercise – Timers.....	4-39
OLE Container .....	4-40
Exercise – OLE Container .....	4-43
Animate .....	4-44
MRegion .....	4-45
StatusBar .....	4-46

ProgressBar .....	4-48
Panel .....	4-49
Splitter .....	4-51
TabControl .....	4-52
Exercise – TabControl .....	4-57
ActiveX Controls .....	4-57
.Net Controls .....	4-58
Adding Menus to Forms .....	4-63
Exercise – Add a Menu to the Inquiry Form .....	4-68
ImageList and ToolBar .....	4-70
Exercise – Add a Toolbar to the Inquiry Form .....	4-75
StatusBar .....	4-76
Exercise – Add a Statusbar to the Inquiry Form .....	4-79
ListView .....	4-80
Exercise – ListView .....	4-87
TreeView .....	4-88
Exercise – TreeView .....	4-92
Adding Forms to Programs .....	4-93
Removing Forms from a Program .....	4-99
Exercise – Adding Forms .....	4-100

## Chapter Five – Principals of GUI Design

Chapter Overview and Objectives .....	5-1
Creating a User Interface .....	5-2
Use Consistency in Your Design .....	5-4
Guide the User through the Interface .....	5-5
Make it Friendly by Giving Good Feedback .....	5-6
Human Factor .....	5-9

## Chapter Six– PL/B Programming

Chapter Overview and Objectives .....	6-1
Designing a Visual PL/B Program.....	6-2
Example of Designing a Visual PL/B Program .....	6-3
Exercise – The Investment Calculator .....	6-4
Coding the Investment Calculator.....	6-6
Variables .....	6-7
Statements .....	6-10
Working with String and Numbers .....	6-11
Exercise – Coding the Investment Calculator .....	6-12
Constants .....	6-13
Exercise – The Math Project.....	6-15

## Chapter Seven– Using the Debugging Tools

Chapter Overview and Objectives .....	7-1
Introduction to Debugging.....	7-2
Debugging Tools.....	7-4
Debugging Techniques .....	7-6
Exercise – Debugging .....	7-14

## Chapter Eight– Data Entry Validation

Chapter Overview and Objectives .....	8-1
The ALERT Instruction.....	8-2
Exercise – The ALERT Instruction .....	8-9
Using ALERT for an About Dialog .....	8-10
Exercise – Use ALERT for an About Dialog .....	8-11
Validating User Input .....	8-12
Exercise – Investment Calculator Validation .....	8-22
Preventing Validation .....	8-23

Exercise – Adding a Clear Button to the Calculator .....	8-25
Validating with Other Controls.....	8-26
Exercise – Create the Employee Data Form.....	8-32
Working with Dates.....	8-34
Exercise – Modify the Employee Data Form.....	8-36

## **Chapter Nine – More Programming Fundamentals**

Chapter Overview and Objectives .....	9-1
Understanding Events in Windows .....	9-2
Exercise – Event Triggering.....	9-4
Functions.....	9-7
Exercise – Add a Function to the Math Project .....	9-11
Passing Arguments to Functions .....	9-12
Collections.....	9-15

## **Chapter Ten – Text File IO**

Chapter Overview and Objectives .....	10-1
Creating a Note Editor .....	10-2
Using the Windows Clipboard.....	10-3
Exercise – The Note Editor.....	10-4
Common Dialogs.....	10-6
Exercise – Opening and Saving Files .....	10-9

## **Chapter Eleven– ISAM Data Access**

Chapter Overview and Objectives .....	11-1
Building the Sales Application.....	11-2
Exercise – Begin the Sales Application.....	11-4
Creating the Customer Form .....	11-7
Exercise – Add the Customer Form .....	11-10

Adding ISAM Access .....	11-13
Modular Design .....	11-14
ISAM Access Routines .....	11-15
Open Routine .....	11-16
Prepare and Close Routines.....	11-17
Read Routine .....	11-18
First and Last Record Routines .....	11-19
Previous and Next Record Routines.....	11-20
New Record Routine .....	11-21
Save Routine.....	11-22
Deletion Routine.....	11-23
Count Routine .....	11-24
Put Routine .....	11-25
Get Routine .....	11-26
Data Verify Routine.....	11-27
Calling Access Routines from Forms .....	11-28
Exercise – Adding ISAM Access to the Customer Form.....	11-30

## **Chapter Twelve– Advanced Data Access Techniques**

Chapter Overview and Objectives .....	12-1
The Order Maintenance Form .....	12-2
Exercise – Create the Order Maintenance Form.....	12-3
Enhancing the Order Form .....	12-6
Exercise – Enhancing the Order Form.....	12-11
Adding Order Detail .....	12-19
Exercise – Adding Order Detail.....	12-28
Order Detail Maintenance.....	12-29
Exercise – Completing the Order Detail Maintenance.....	12-44



## Chapter Thirteen– Adding AAM Access

Chapter Overview and Objectives .....	13-1
The Customer Maintenance Form .....	13-2
The Customer Definition File .....	13-3
The Main Program .....	13-4
The Customer Form .....	13-5
Customer Access Routines.....	13-6
Exercise – Adding AAM Access .....	13-14

## Chapter Fourteen– Advanced Print

Chapter Overview and Objectives .....	14-1
Introduction to Advanced Printing .....	14-2
PRTOPEN.....	14-3
PRTCLOSE .....	14-7
PRTPAGE .....	14-10
PRTPLAY Example.....	14-11
Exercise – Advanced Printing .....	14-14

## Appendix A – Translation Table

Translation Table .....	A-1
-------------------------	-----

## Appendix B – PL/B Utilities

PL/B Utilities .....	B-1
BLOKEDIT .....	B-2
BUILD .....	B-4
OBJMATCH.....	B-5
REFORMAT .....	B-6
SUNAAMDY.....	B-7
SUNINDEX.....	B-12

SUNLIST .....	B-17
SUNSCAN.....	B-21
SUNSORT.....	B-22
TXTMATCH.....	B-25

# **COURSE OBJECTIVES**

- Understand the basic concepts of Visual PL/B programming.
- Become familiar with the Integrated Development Environment.
- Learn how to design, create, and debug Visual PL/B applications.
- Understand the objects that compose the Standard Tool Set.
- Gain insights into good GUI programming practices.
- Understand debugging techniques available within Visual PL/B.
- Learn how to perform data entry validation in a GUI environment.
- Become exposed to PL/B language variables and structures.
- Understand PL/B text file handling.
- Learn the types of data access available.
- Implement a data access structure.
- Gain exposure to advanced data access techniques

## **CHAPTER ONE**

### ***INTRODUCTION TO VISUAL PL/B***

## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter will discuss the features of Sunbelt Visual PL/B, the different types of applications you can create with Visual PL/B, and some terminology.

Upon completion of this chapter, you will be able to:

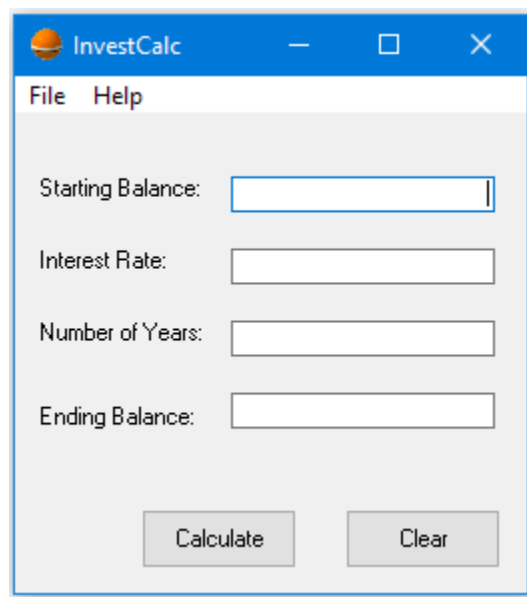
- Identify the features of Visual PL/B
- Understand the type of applications you can build with Visual PL/B
- Understand some of Visual PL/B's terminology.

## WHAT IS VISUAL PL/B?

Visual PL/B is a programming suite that allows you to develop applications in a **Graphical User Interface (GUI)** environment like **Windows** or a **Web Browser**.

A Graphical User Interface (GUI) enables you to select commands and process data by pointing at and clicking on pictures, symbols (icons), data, and buttons.

Windows is an operating system that uses graphical objects as an interface between the user and the application.



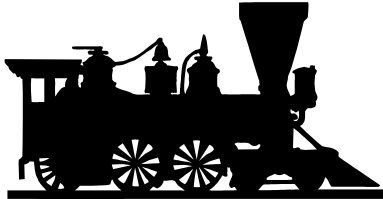
**Example Window**

A window can contain:

- A **Title Bar** that names the application.
- **Minimize**, **Maximize** and **Close** buttons in the upper right corner to allow minimizing to an icon, maximizing to occupy the whole screen, or closing the application.
- A **Menu Bar** with pull-down menus that list options for the user.
- Various graphical controls (i.e., pictures, labels, buttons, text boxes) that provide the user with a way to communicate with the application.

## **WHAT IS VISUAL PL/B? – continued**

Visual PL/B applications are said to be event-driven applications. That is to say that they do not work in a linear fashion like traditional procedural applications. Code remains idle until called upon in response to a particular event such as a mouse click, key press or system-timed event.



An analogy might be that a linear program is much like a train ride. You get on the train, you go where the train takes you, it stops, and you get off. You are not in control of where the train goes.

The event-driven program is much like driving a car. You start the car, but it does nothing until you make it happen by putting the car in gear and depressing the accelerator. You are in total control of where the car goes.

Visual PL/B can only be run on 32-bit Windows environments such as Windows 7, 10 and 11.



Visual PL/B is an object-oriented, Rapid Application Development (RAD) programming tool.

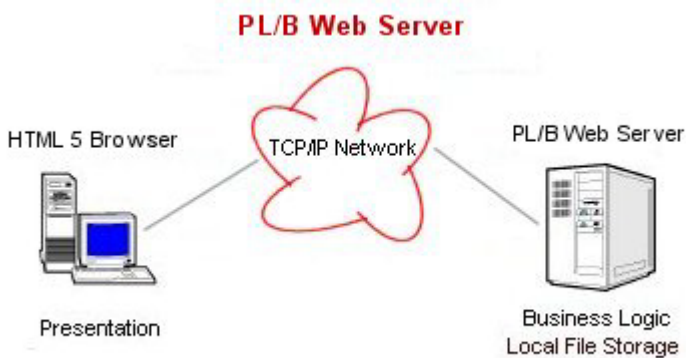
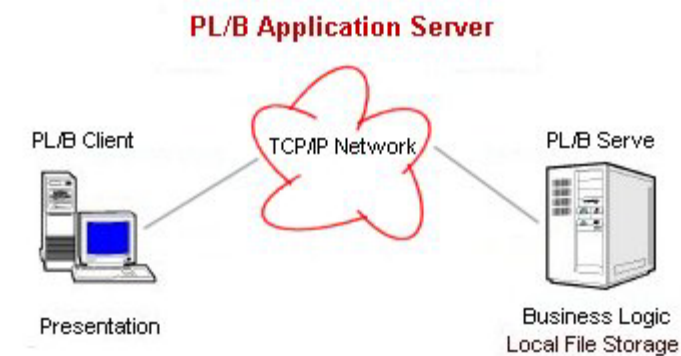
Visual PL/B has an Integrated Development Environment (IDE) that allows you to design your entire user interface, code the functionality required for your application, test, and debug your application.

## Visual PL/B Application Types

Advanced features in Visual PL/B and other products from Sunbelt allow you to create fast, high-performance **client/server applications** and **components** that can be used by the Application Server or the Web Server.

Your Visual PL/B client/server application is responsible for

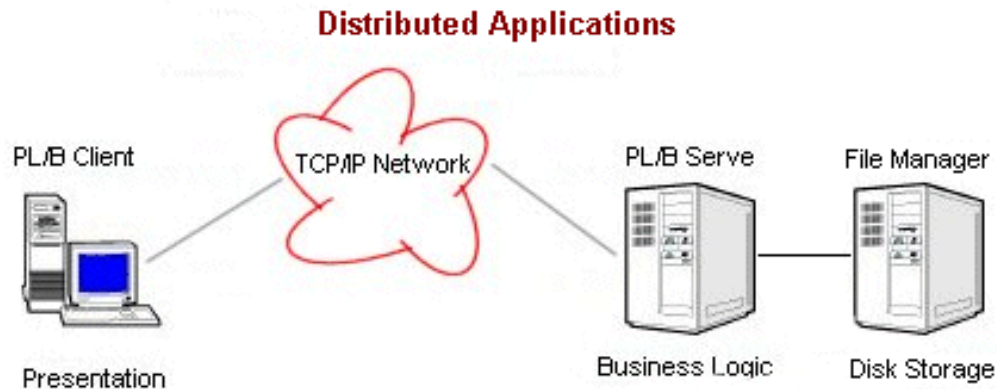
- Managing the screen display for the user
- Interacting with the user
- Validating user input
- Requesting data from the server
- Processing the data returned from the server





You can also build **distributed** (n-tier) applications with Visual PL/B and other Sunbelt Products.

In this architecture, an **application server** component stands between the clients and the server.

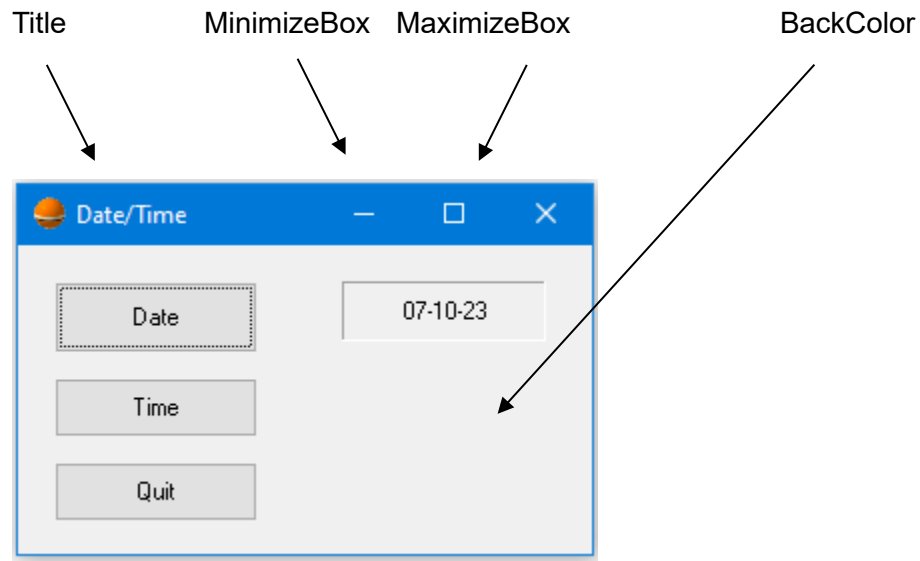


The application server can perform some of the processes of the client and/or the server, such as handling requests for data and processing data.

## VISUAL PL/B TERMINOLOGY

Before you can begin learning how to create applications in Visual PL/B, you must understand some terminology.

Almost everything you do with Visual PL/B centers around **objects**. All objects are created from **classes**. A class defines what an object looks like and what it can do. Some examples of objects in Visual PL/B are windows, command buttons, editable text boxes, and static text.



The most important thing to remember is that objects all possess at least one of the following:

- **Properties**
- **Methods**
- **Events**

**Properties** are the characteristics of an object that control how it looks and how it behaves. For example, a window has properties such as:

- **BackColor** (describes the background color of a window)
- **Title** (defines the text that is displayed in the Title Bar)
- **MinimizeBox** and **MaximizeBox** (controls the window's behavior)

**Methods** are commands that can be performed with or on an object. For example, a window can respond to the Show method, which makes the form visible.

**Events** are triggered when the user performs an action on an object. For example, the click event is triggered with the use clicks the mouse pointer on a Command Button. There are also system events that can occur.

**Note**

All objects in Visual PL/B have at least one property, event, or method.

## **CHAPTER TWO**

### ***EXPLORING THE VISUAL PL/B ENVIRONMENT***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter describes the Integrated Development Environment in Visual PL/B.

Upon completion of this chapter, you will be able to:

- Understand the elements of the Visual PL/B Integrated Development Environment (IDE)
- Identify the primary windows in the Visual PL/B design environment
- Use the options available to modify your design environment

## **THE VISUAL PL/B DESIGN ENVIRONMENT**

To start Visual PL/B, select Sunbelt IDE from the Program Group for Sunbelt Visual PL/B. To do this, select Start, then Programs, then look for Sunbelt Visual PL/B group in the Windows program menu. There may also be a shortcut on your Windows desktop that you can double-click. The icon below is an example.



Sunbelt IDE Ink

Sunbelt IDE Studio

When you start Visual PL/B, you may see the following screen:



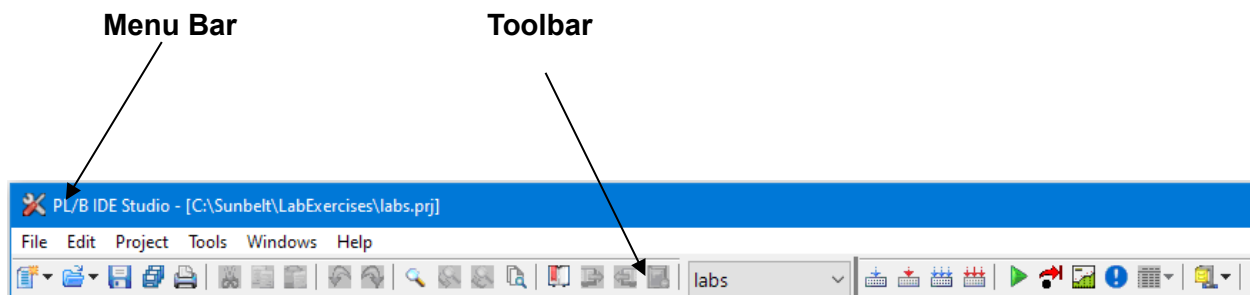
## **THE VISUAL PL/B DESIGN ENVIRONMENT - continued**

Traditional application development involves three steps: writing code, compiling, and testing. The working environment in Visual PL/B is referred to as the Integrated Development Environment because it combines many different functions such as design, editing, coding, compiling, and debugging within one common environment.

Several windows are used in the design of a Visual PL/B application. Let's examine those first.

The Main Window contains the:

- Menu Bar with six drop-down menus
- Toolbar with many short cut icons for frequently used menu commands



Tool Tips are available for each of the enabled icons on the toolbar and can be viewed by passing the mouse pointer over an icon.



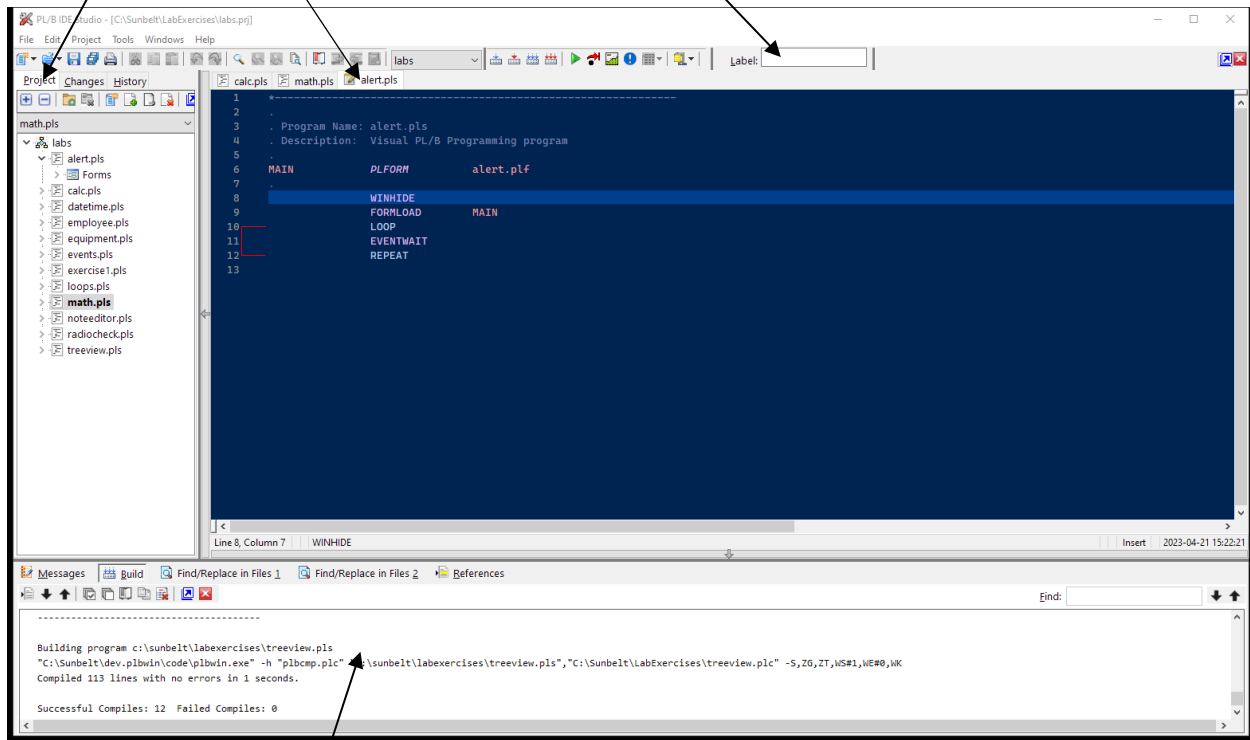
## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

The IDE has several windows that manage the development of an application.

Project Tab

Source Code Editor

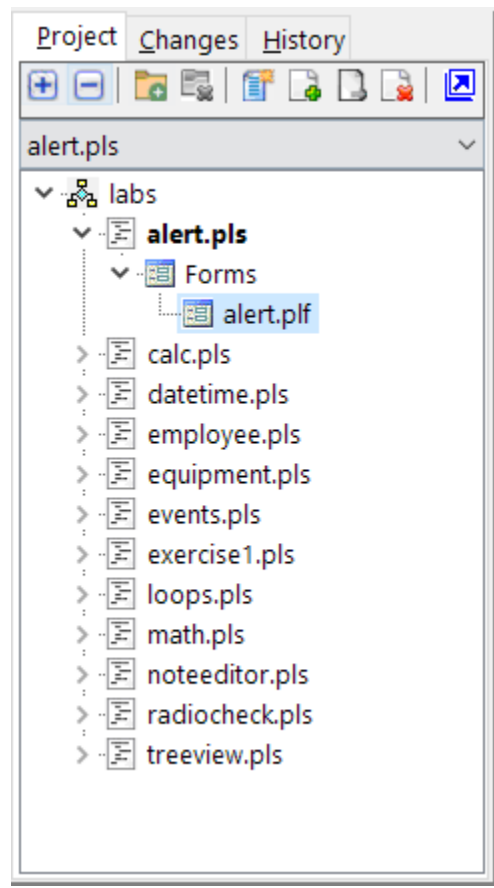
Browse Code Label Field



The build output window

## **THE VISUAL PL/B DESIGN ENVIRONMENT - continued**

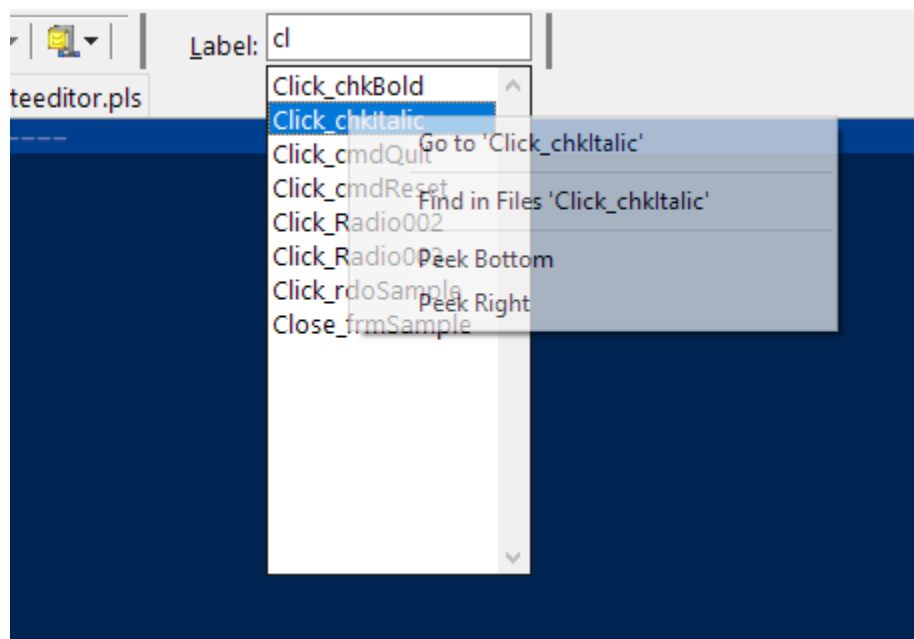
The Project tab contains a list of all the source files in the current project. The term project simply refers to a collection of Visual PL/B programs. A source file creates one application program. Each source file may have other source files (includes) or form files (PLFORMs) as dependencies. Each new project starts with no source files. We will discuss managing projects and project components in a subsequent chapter. Double-clicking a file will open the appropriate editor for the source file.



**The Project Tab**

## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

The Browse Code Label Field is located after the toolbar. This is a handy tool used by programmers to find the location of code labels. The Go to option will open the appropriate editor for the source file and position to the line containing the label. The Find in Files option will locate all the places the label is referenced.



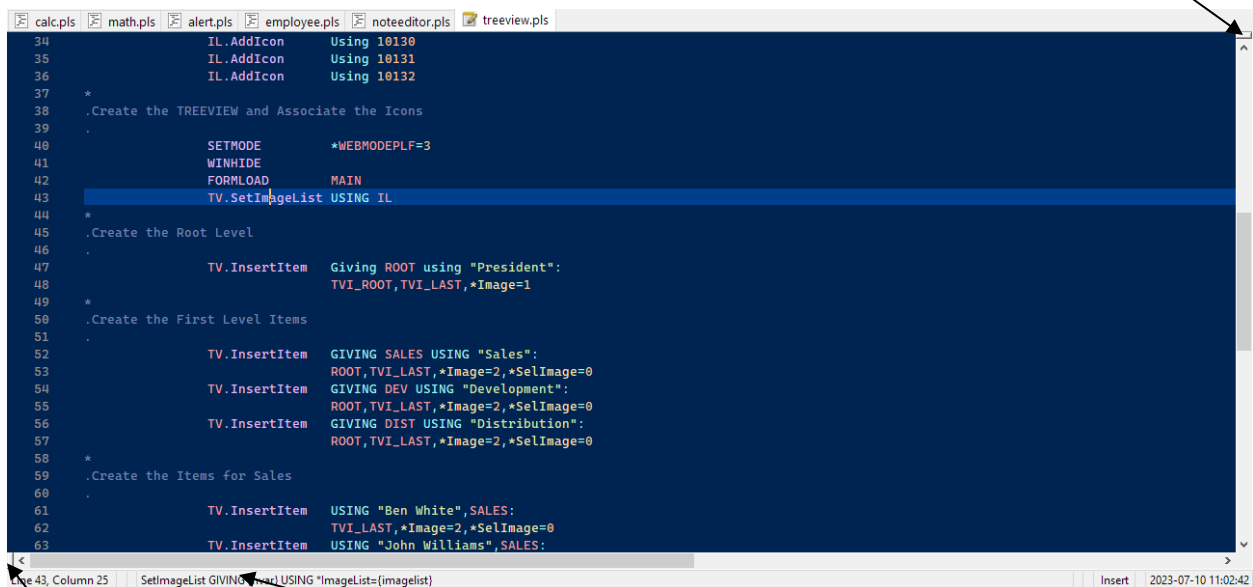
**The Browse Code Label Field**

## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

The source editor included in the Sunbelt IDE Studio allows multiple files to be open concurrently. Files are selected using a tab at the top of the editing window. The editor supports syntax coloring, horizontal and vertical splits, code completion, and many more features. The IDE also has a mechanism to allow replacement of the supplied editor with one of the users own choosing.

File Selection Tab

Horizontal Split Tool

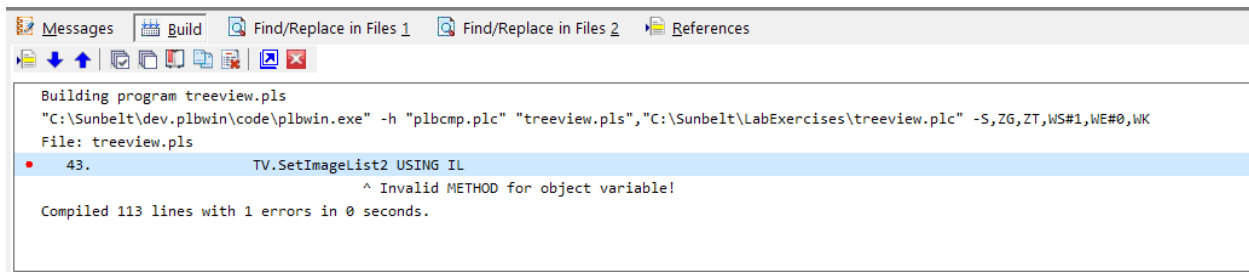


Vertical Split Tool

Syntax Information

## **THE VISUAL PL/B DESIGN ENVIRONMENT - continued**

The Build Output Window displays operational messages. The result of application builds will appear here. Should the compiler encounter an error, it will be displayed. Users often adjust the size of the window to view more messages.



**Build Window showing a compiler error**

### **NOTE**

Double-clicking an error message will open the appropriate source file in the editor and position to the line containing the error.

## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

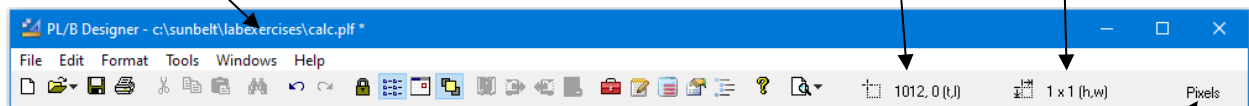
An integral part of the Sunbelt IDE Studio is the Form Designer. The designer allows creation of GUI forms that are then included in application programs. GUI applications will normally contain numerous forms.

The main menu and toolbar of the Form Designer provides some insight into its capabilities. There are provisions for opening, creating and saving forms. Normal editing functions such as cut, copy, paste, and delete are also supported.

The name of the form being edited is shown in the title bar. An asterisk appears if the form has been modified but not saved. Additional information includes the position and size of the selected form object and the form's drawing units.

Form Name

Selected Object's Position and Size



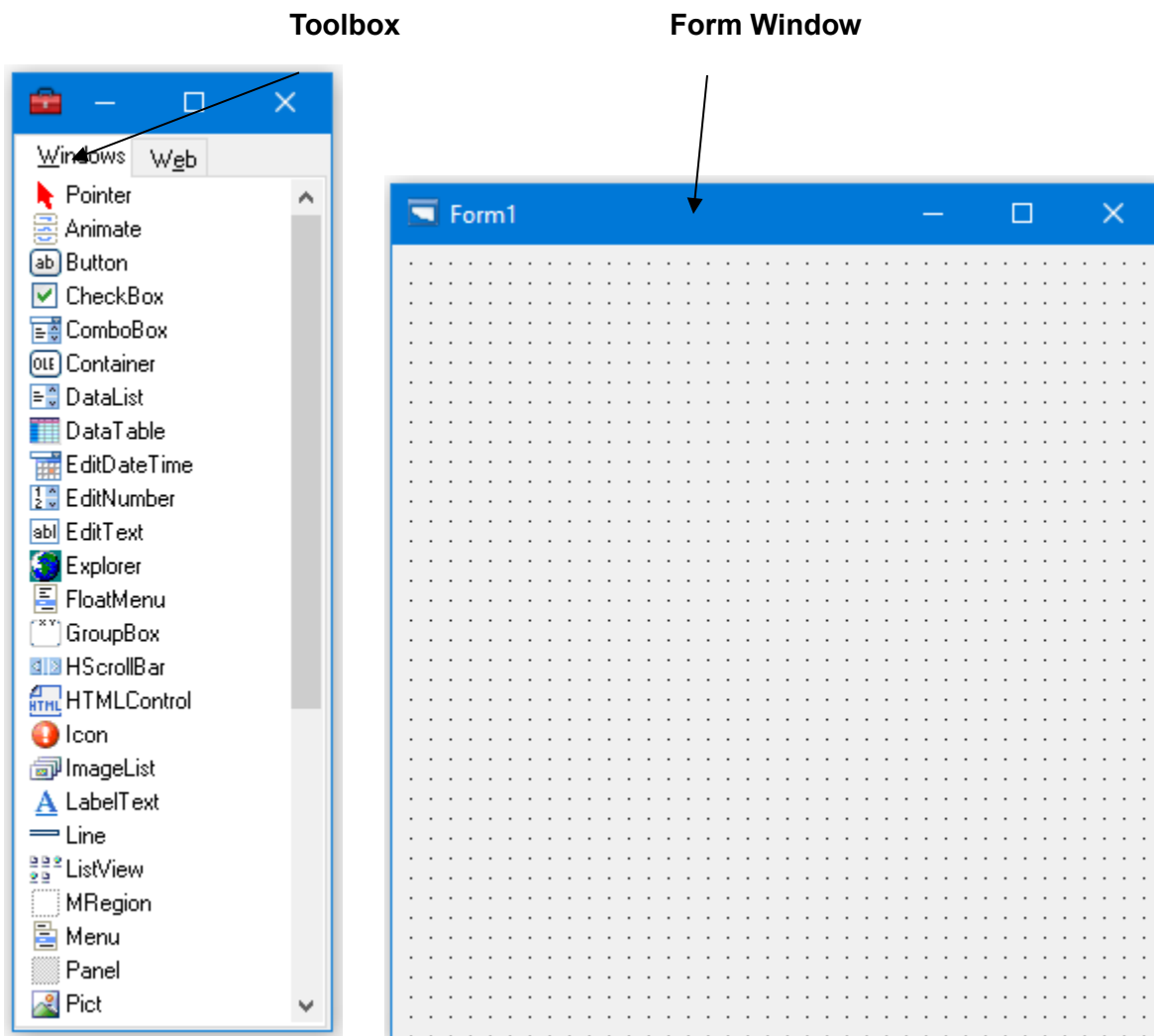
Drawing Units

## **THE VISUAL PL/B DESIGN ENVIRONMENT - continued**

The Form Window and Toolbox are used together with the Form Designer to create the user interface.

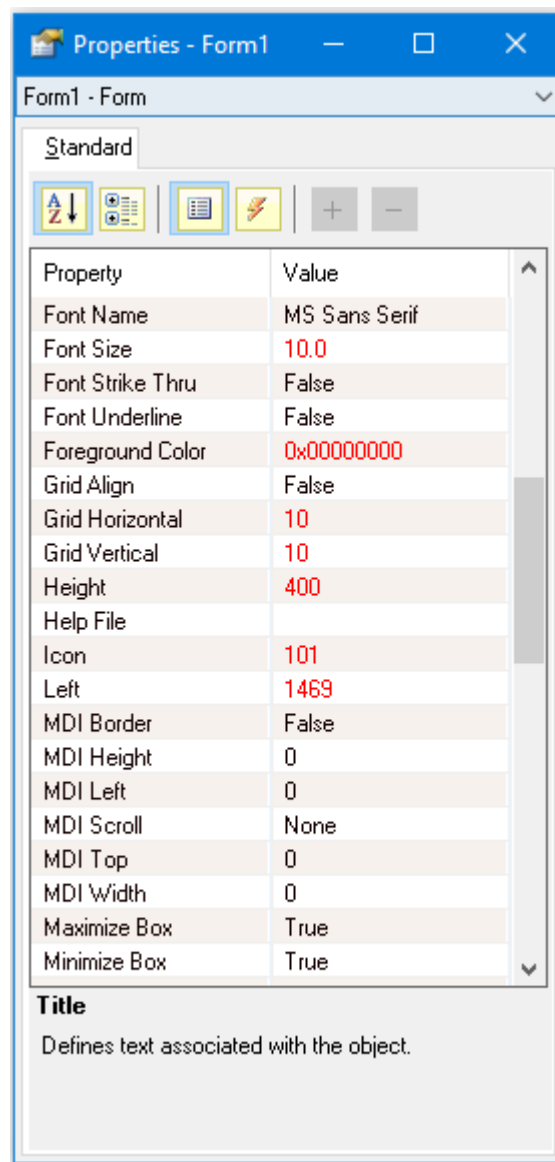
The Form window is the foundation for your user interface. You normally start each new project with one blank form and you can add as many others as you need while you are developing the application. Each form corresponds to a window or dialog box in the project.

The Toolbox provides the controls you need such as command buttons, image boxes, and scroll bars that you place on your form. As you pass the mouse pointer over the icons in the Toolbox, a tool tip will appear indicating the type of tool the icon represents. We will explore many of the controls in the Toolbox in subsequent chapters.



## **THE VISUAL PL/B DESIGN ENVIRONMENT - continued**

The Properties window displays and sets the attributes of the controls drawn on the form. There are default properties set for each object by Visual PL/B when you start a new project. Use the Properties window to change the default settings. Property settings control each object's appearance and behavior. We will explore the Property window in detail in a subsequent chapter.

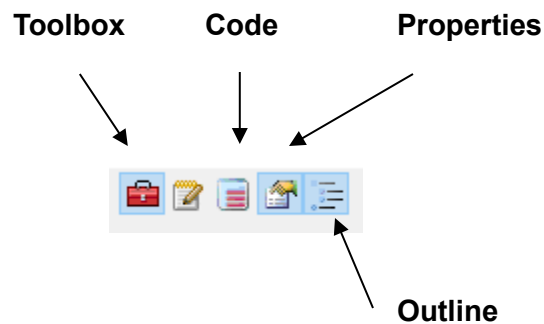


**Property Window**

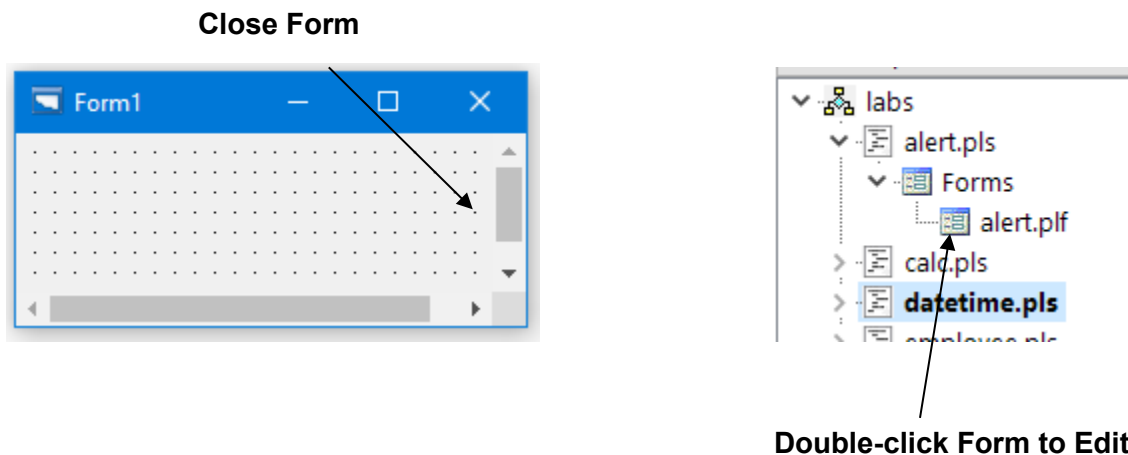


## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

If you close the Toolbox, Code, or Properties window within the Form Designer, you can show them again by clicking on their icons in the Form Designer's toolbar

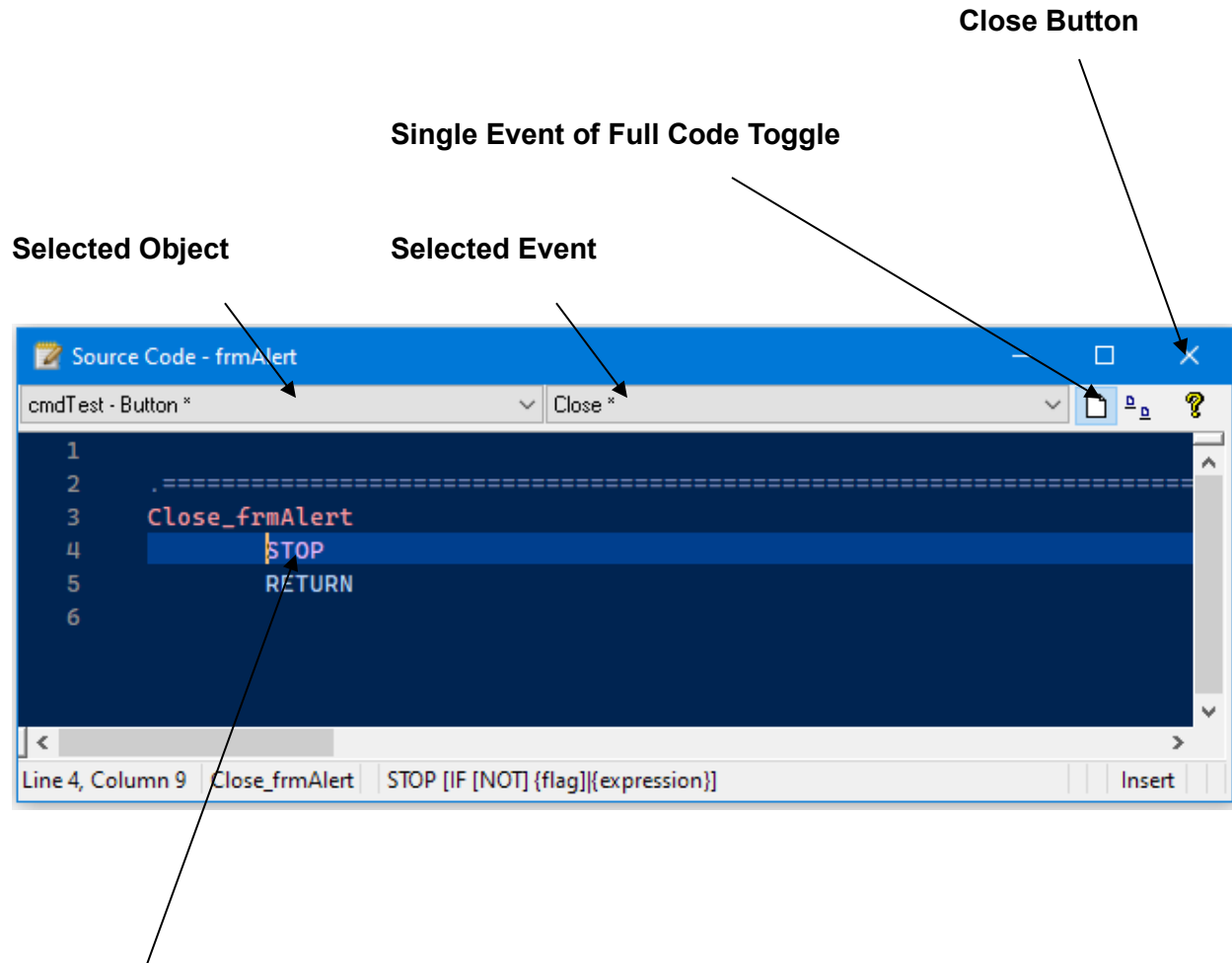


If you close the Form Window, you can re-open it by double-clicking the form name in the Project window.



## THE VISUAL PL/B DESIGN ENVIRONMENT - continued

The Code window can be opened by double-clicking anywhere within the form. It is closed by clicking the window's close button.



### Event Routine Code

The editor employed by the Form Designer is the same editor as used in the IDE. All features such as syntax coloring, code completion, horizontal and vertical splits, and language help are available within the code window.

## **CHAPTER THREE**

### ***CREATING A VISUAL PL/B APPLICATION***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter describes the basic steps of creating a Visual PL/B application and explains how to begin using Visual PL/B to create a graphical user interface.

Upon completion of this chapter, you will be able to:

- Define the basic steps involved in creating a Visual PL/B application.
- Demonstrate the several methods of adding controls to a Visual PL/B form and how to use the designer to move, size, and align controls.
- Identify the types of property settings and demonstrate changing the default properties at design time.
- Utilize the code window to add code to a Visual PL/B object.
- Use the IDE to save and compile a Visual PL/B application.
- Demonstrate how to run a Visual PL/B application.

## **VISUAL PL/B APPLICATION DESIGN STEPS**

There are twelve basic steps in the physical design of a Visual PL/B application:

1. Define the user interface – that is, consider the form(s) required for the application.
2. Create a project that will contain the application.
3. Create the source program for the application.
4. Write the source program's code.
5. Save the source program.
6. Create a form.
7. Draw the controls (command buttons, text boxes, etc.) on the form.
8. Set the properties such as name, colors, and size as required for the objects.
9. Write code so that the application responds to the user.
10. Save the form.
11. Build the application.
12. Run the program.

These steps may be repeated often during your application development stage.

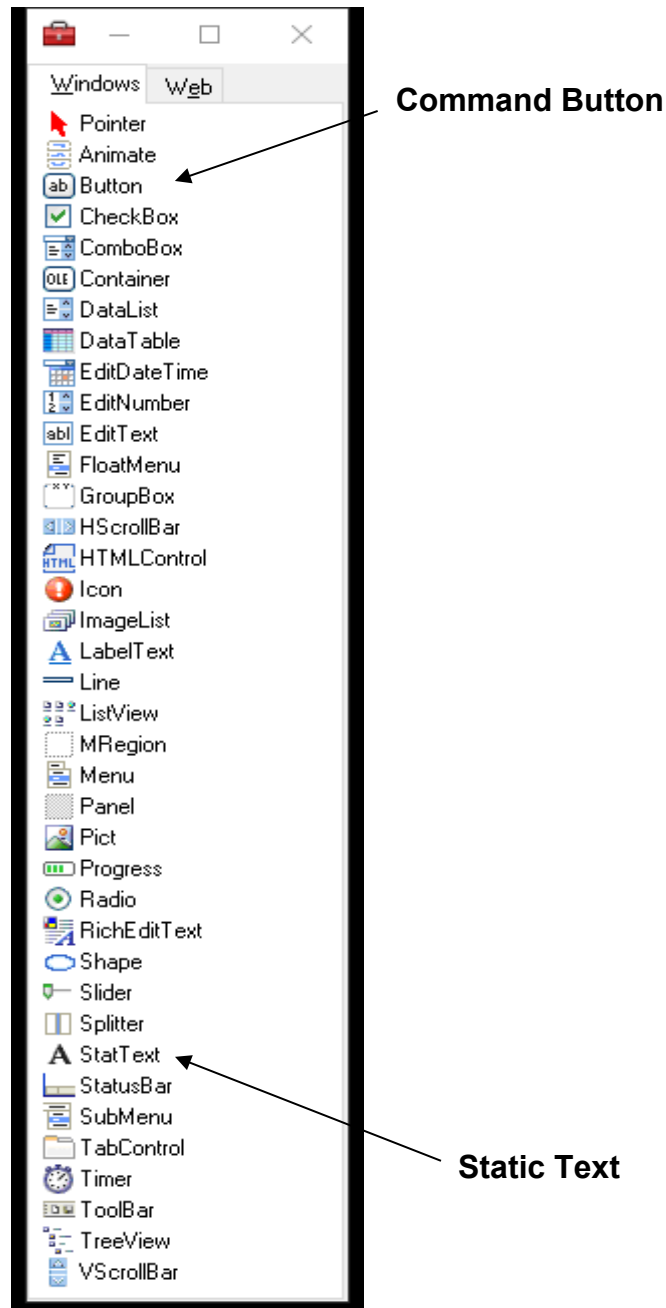
We will demonstrate these steps in this chapter using a simple project so that you will have the experience of each of the steps.

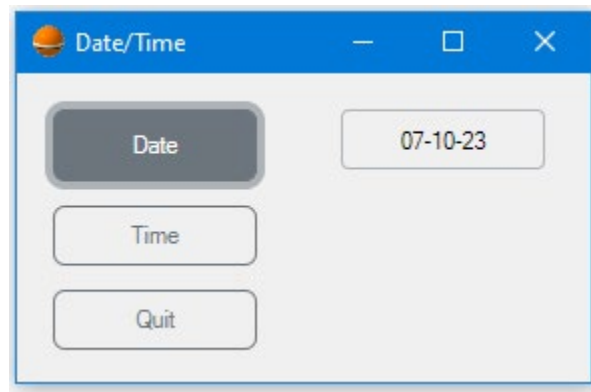
## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 1 – Design the User Interface**

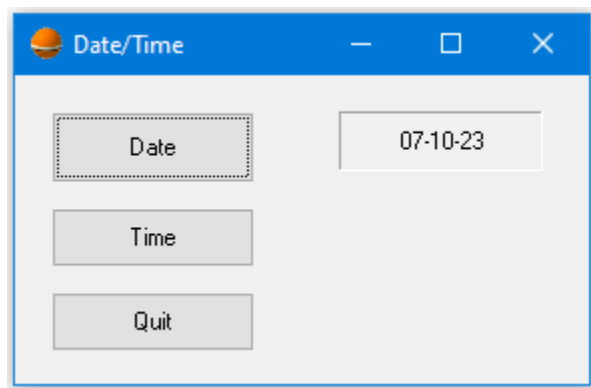
Let's create an application that allows you to click on different buttons to display either the date or time in a static text object on the form and a button that will allow you to end the application.

This should require one form, three buttons, and a static text object.





**WebView Sample Form**



**Win32 Sample Form**



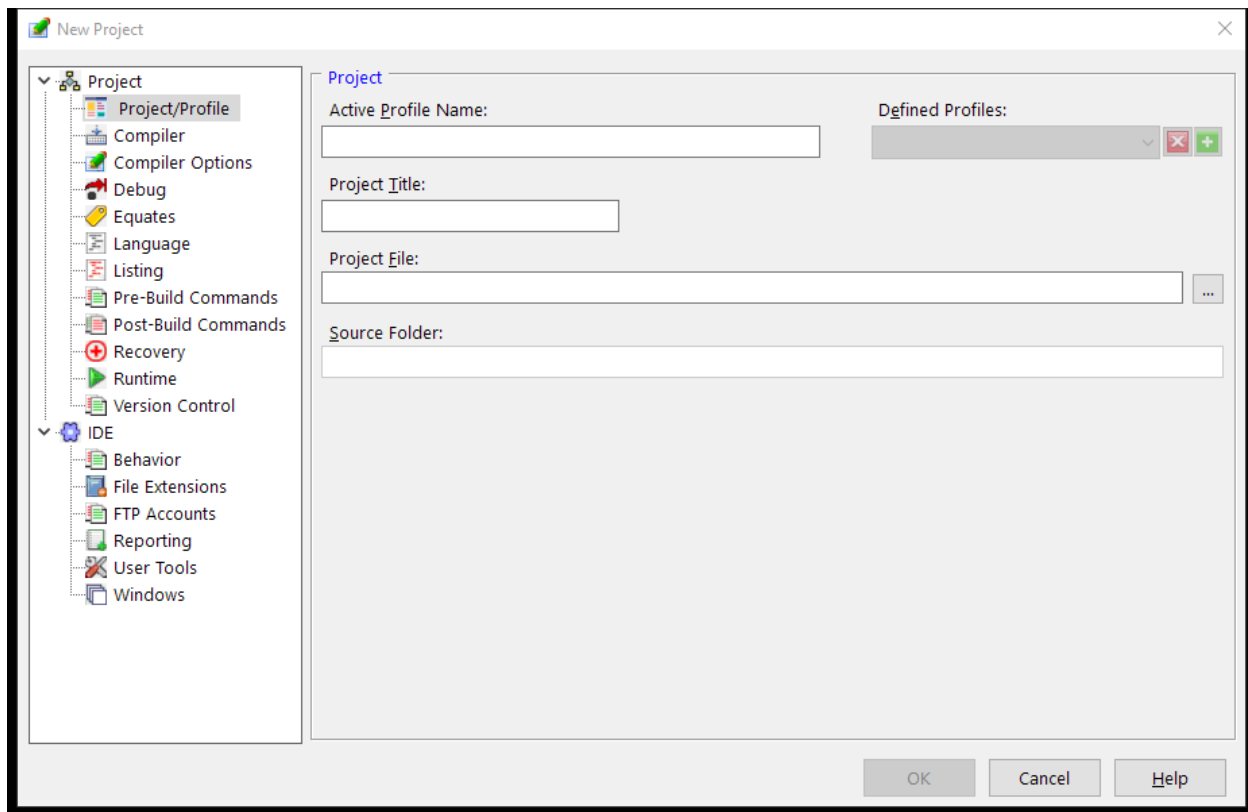
## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 2 – Create the Project**

To begin the application, we'll need a project.

Start Visual PL/B from the desktop or the Windows Start Menu.

From the File menu, select "New Project". The following dialog is displayed:



**New Project Dialog**

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 2 – Create the Project - continued**

For the "Name of Project", enter "Exercises".

In the "Project File Name/Location" text box, enter "c:\myname\Exercises.ppj" where "myname" is your name or the name of a directory in which the exercise files will be stored. Alternatively, you may use the browse button to locate a folder. The folder must already exist.

In the "Working/Source Directory" text box, enter "c:\myname\" where "myname" is the same folder name specified in step 4.

Click OK. The project will be created.

The screenshot shows a 'Project' dialog box with the following fields and values:

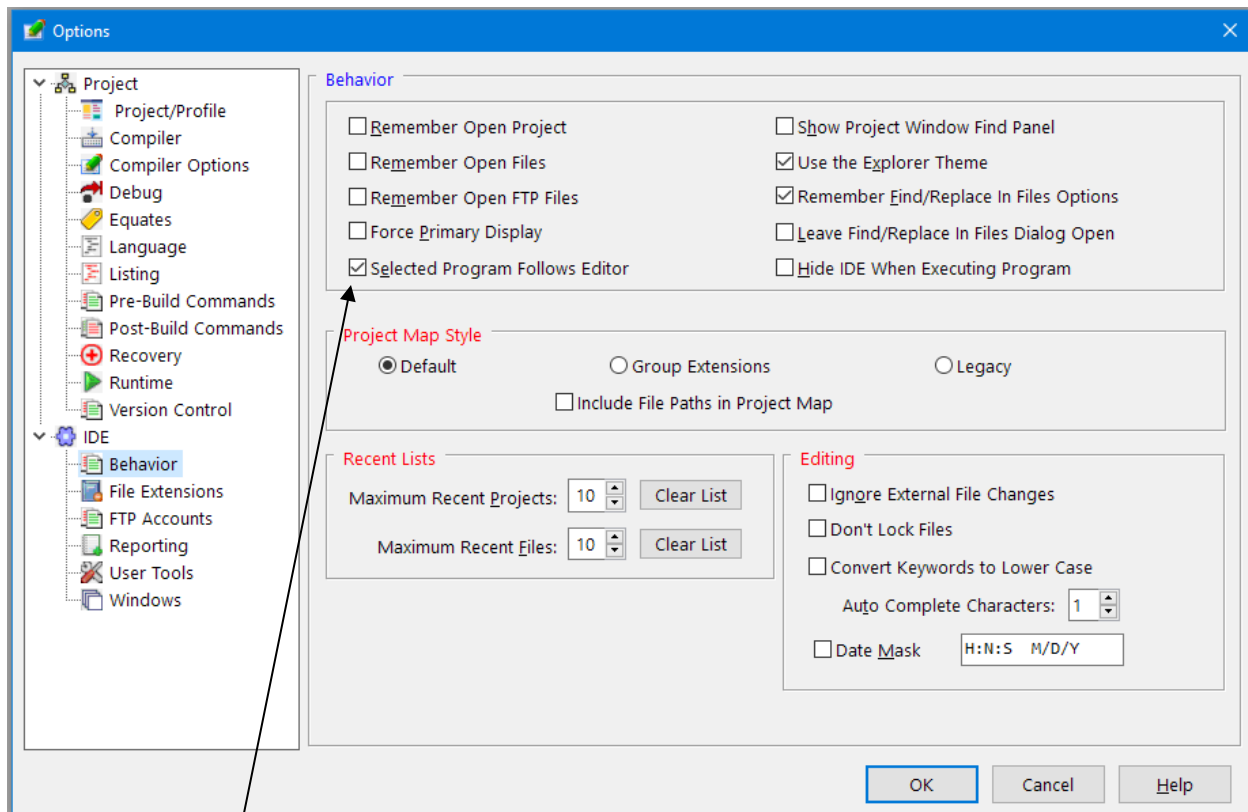
- Active Profile Name:** Exercises
- Defined Profiles:** (empty dropdown menu with X and + buttons)
- Project Title:** Exercises
- Project File:** C:\Sunbelt\sample\Exercises.prj
- Source Folder:** C:\Sunbelt\sample\

Buttons at the bottom: OK, Cancel, Help.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 2 – Create the Project - continued

1. While setting options, one final IDE option that should be set is the Selected Program Follows Editor on the IDE options dialog.



IDE Options Dialog

Enable this option

**EXERCISE – CREATE A PROJECT**

1. If you have not done so already, create a project as described on the previous pages.
2. Pay special attention to the items you are requested to supply.
3. If the working directory does not exist, you may create it using the File Explorer.

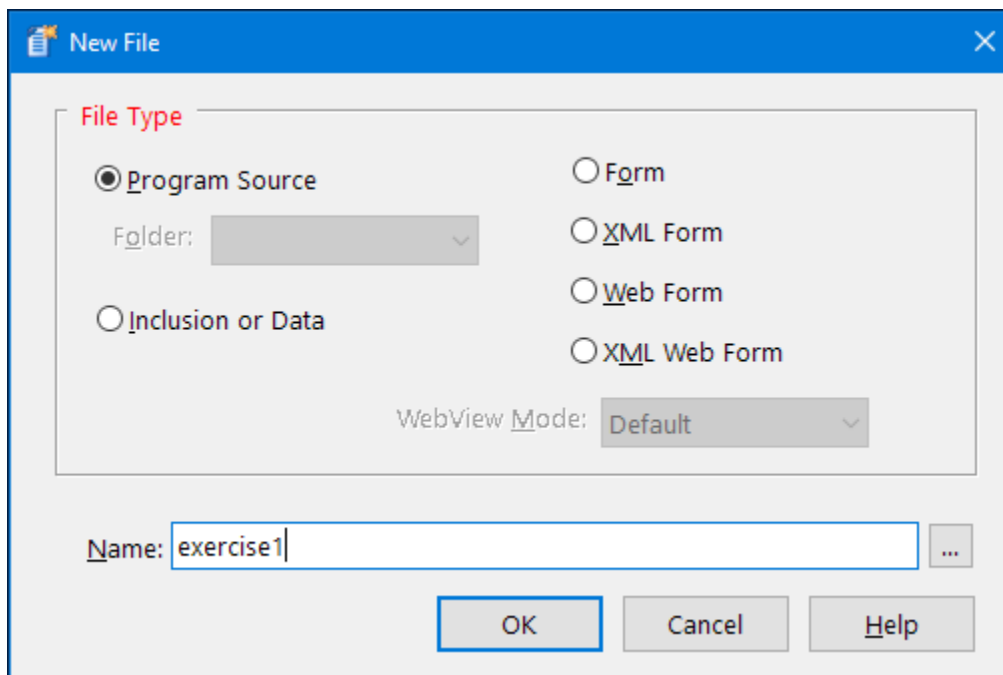
## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 3 – Create the Source Program**

Once the project has been created, the next step is to create the source program.

To create the source program:

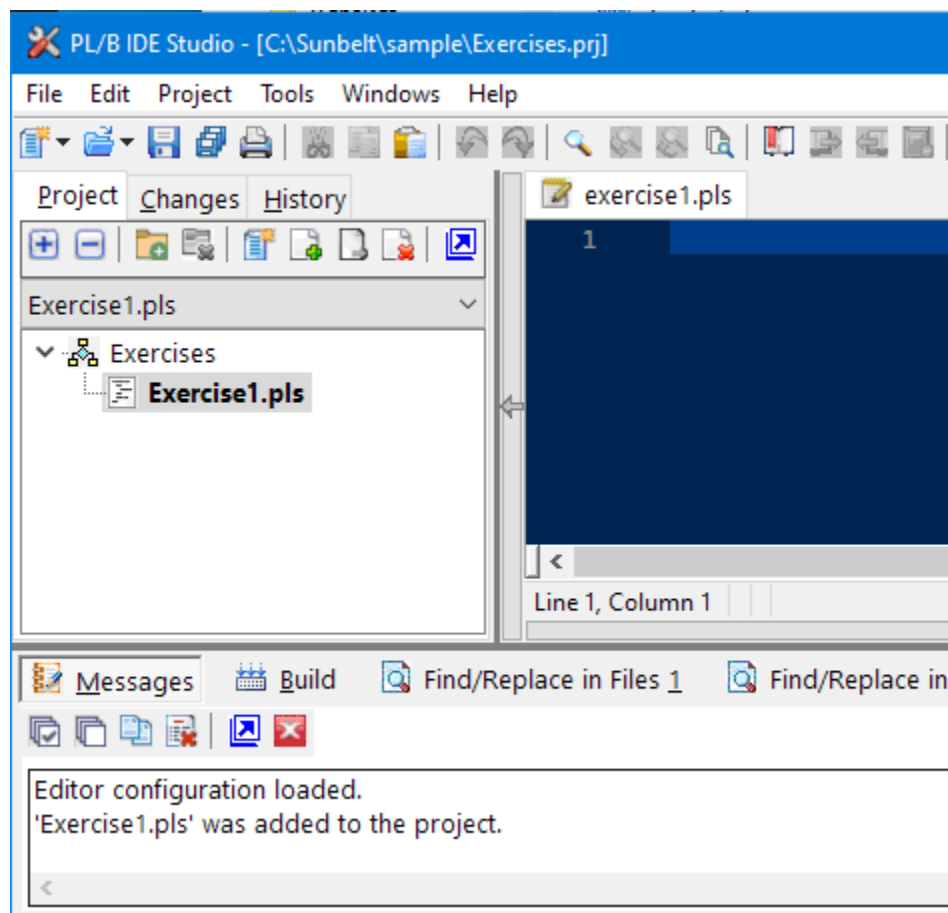
1. Click "New File..." from the File menu.
2. Enter "exercise1" for the file name. The file extension will default to "pls".
3. Click the "OK" button to add the new source file to the project.



## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 3 – Create the Source Program**

Once the source program is created, the IDE appears as follows. Note that the file has been added to the Project Tab and the name is displayed in the heading. The editing window is opened, and the cursor placed at the top of the window ready for source code input.



## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 4 – Write the Source Program Code**

In this exercise, we will use the smallest possible source program as the basis of our application. This program will simply load the form we have yet to design and dispatch events.

You will notice that after the source program was added to the project, an editing window appeared, and the cursor is blinking in it. That signifies that the code editor is ready for input. Add the following lines to the code window.

```
MAIN          PLFORM      EXERCISE1.PLf
.
              FORMLOAD    MAIN
.
              LOOP
              EVENTWAIT
              REPEAT
```

PL/B is not case sensitive so you may use upper, lower, or mixed case as you prefer.

The first line containing the PLFORM verb identifies a form file that will be included in your program. In the program code, it is referred to using the "MAIN" label.

The second and fourth lines are comment lines and just make the source file a little more readable.

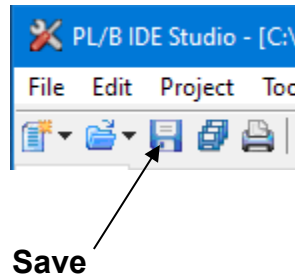
The third line directs the program to load the form. It will also become visible to the user at that point.

Lines five through seven create an endless loop in which the program waits for events and dispatches them accordingly.

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 5 – Save Your Source Program**

Before proceeding, it would be wise to save the newly created source program. To save the program, click Save in the IDE toolbar or click Save under the File menu.



Since the file name and directory were specified during the add source file procedure, no input is required. To save the source file with a different name or in another directory use the Save As function in the File menu.



## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 6 – Create the Form**

During the design phase of our program, we determined that we would need one form. With the source program in place, the next step is to create the form as follows:

1. Click New File..." from the File menu. Select the Form option, enter "exercise1" for the file name, and click the "OK" button. The Form Designer will begin execution and the Designer's main window will be displayed.
2. The Designer will create the exercise1.plf file, and the Form Design window will appear. Other windows such as the Toolbox, Code Window, and Property Window may also appear.
3. The Designer will remember the size and position of all windows. To move a window, press the left mouse button while the pointer is in the Title Bar area of the window. Drag the window to its new position and release the left mouse button. The final design position of the form is the default starting position during program execution unless overridden by the form's WindowPos property.
4. The Designer windows are resized in a similar fashion by pressing the left mouse button on an edge of the window and dragging it to a new position. Releasing the button sets the window size. The corners of the windows allow resizing in two directions. The mouse cursor changes as it is positioned over the edges or corners of the window to indicate the direction in which the window may be resized.
5. Some users also prefer the main window of the Designer to be full screen. This prevents any background images from showing.
6. Once you have the various windows sized and in the desired locations, Click "Tools" in the menu and select "Options". Click on the Windows item in the left pane and then set the checkmark for "Use selected form's size and position as default". Click "OK" to exit the options screen.

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 7 – Drawing Controls**

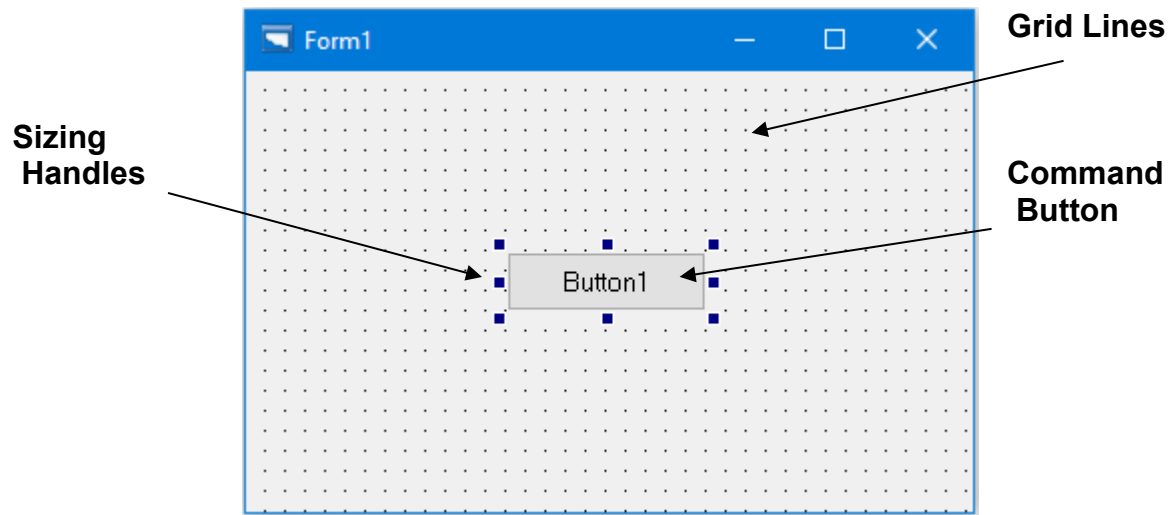
With our form created, we are now ready to place controls on it. There are several methods of placing controls on a form:

1. Double-click a tool in the Toolbox and Visual PL/B creates the object with a default size in the center of the form.
2. Click the tool in the toolbox and then move the mouse pointer to the form window. Position the arrow to the upper left corner of the desired location for the object. Press and hold the left mouse button and drag the mouse to the right and down. The cursor changes to a crosshair dragging to indicate that you are in drawing mode. When you release the mouse button, Visual PL/B will draw the control inside the rectangle you delineated with the drag operation.
3. Multiple objects of the same type and default size may be created by holding the Alt Key down while selecting a tool. Once the tool is selected, the Alt Key may be released. Click on the design form will create objects of the type selected with the default size. You must click the Arrow tool to end the drawing process.
4. Multiple objects of the same type and varying sizes may be created by holding the Ctrl Key down while selecting a tool. Once the tool is selected, the Ctrl Key may be released. Click on the design form and drag to an ending location to create objects of the type selected. You must click the Arrow tool to end the drawing process.
5. Tools may be dragged and dropped from the toolbox to the design form. The objects will be created using the default size.

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 7 – Drawing Controls - continued**

To **move** a control, you have drawn on a form, click the object. The object is now displayed with sizing handles (black boxes) around its perimeter. With the pointer inside the control, press and hold the left mouse button. Drag the object to the desired location on the form and release the mouse button.



Objects may also be moved using the following keyboard methods:

1. The arrow keys will move the object in the direction indicated on the key. If Align To Grid is enabled, the object moves based on the grid unit definition.

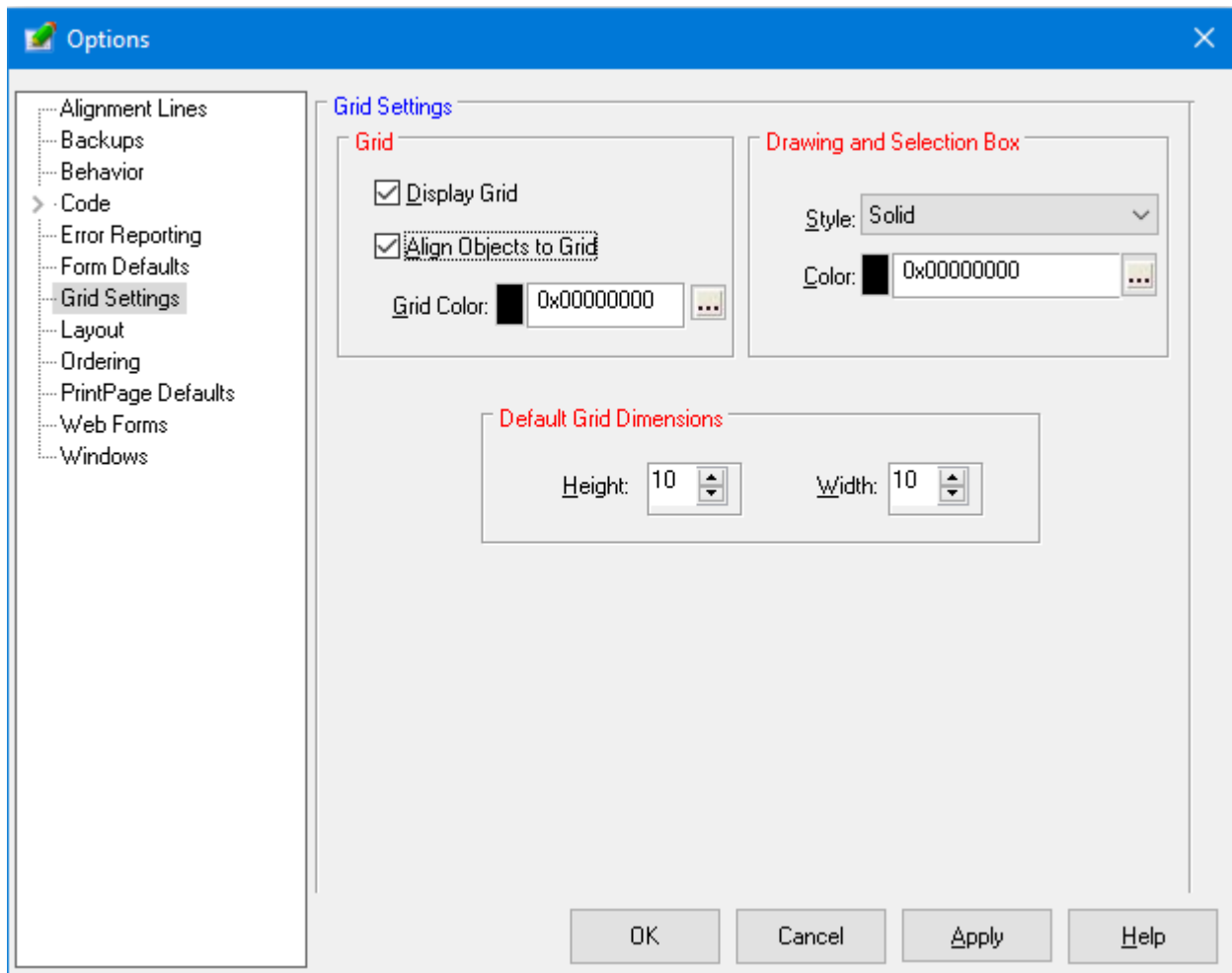
Holding the Ctrl Key while pressing the arrow keys will move the object in the direction indicated on the key one position.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 7 – Drawing Controls – continued

You can **align** controls to the grid as you are creating the interface. The grid is invisible at run time.

To adjust the size of the grid or disable it, choose the "Options" item in the Tools menu and then select "Grid Settings".



The Align To Grid option may also be toggled with a Toolbar button.



Align to Grid

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 7 – Drawing Controls – continued**

To **resize** a control, click the object so that it is selected, and the sizing handles appear on the border of the control. Move the cursor over the appropriate sizing handle until it becomes a double arrow and then drag the handle until the object is the desired size.

Controls may also be resized using the keyboard. After selecting the control, hold the Shift key while using the arrow keys to change the control's size. The size is changed based on the current grid settings if Align To Grid is enabled.

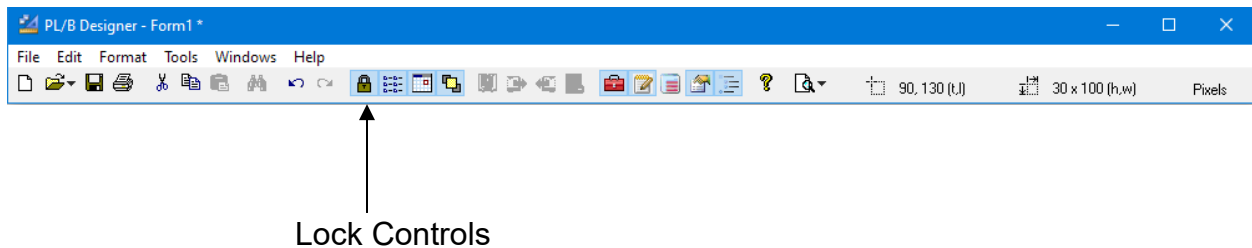
To **delete** a control, begin by clicking the object to select it. Once the sizing handles are displayed around the object, press the Delete key on the keyboard or choose "Delete" from the Edit or shortcut menu.

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

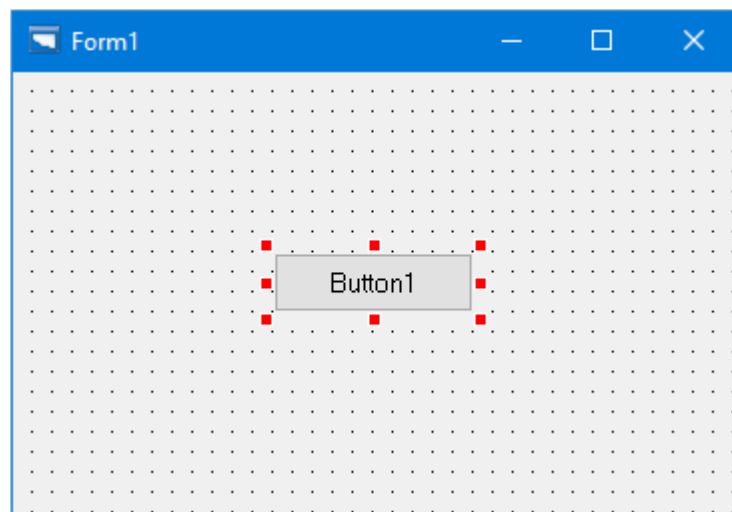
### **STEP 7 – Drawing Controls – continued**

Once you have placed controls on the form in the appropriate position, you can lock them in place by selecting the "Lock Controls" option from the Format menu. This prevents accidental movement of the controls via the mouse. Note that controls may still be moved using the keyboard and they may still be deleted. Only movement and sizing using the mouse is prevented.

Alternatively, you can Lock Controls using the toolbar button.



While locked, the sizing handles for selected objects will appear red rather than black.



**A Locked Control**

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 7 – Drawing Controls – continued**

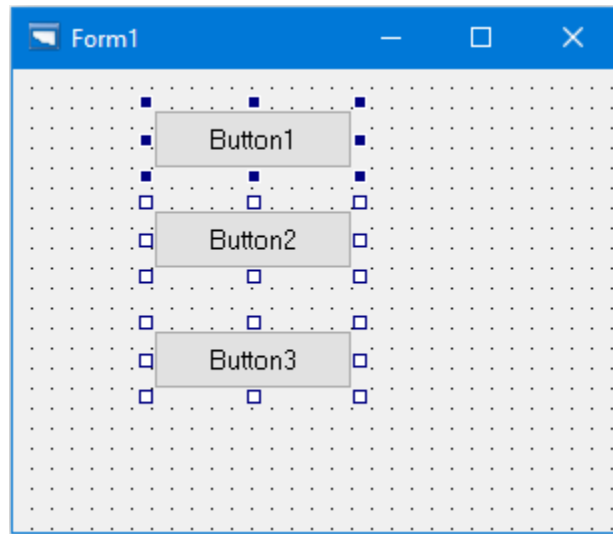
To **select multiple** controls on a form, click the left mouse button outside and above the controls desired and drag the mouse pointer through the objects. Upon releasing the mouse button any object contained within or touched by the defined area will be selected.

An alternate method of selecting multiple objects is performed using the keyboard. Select the first object by clicking it as you would normally. Press and hold the **Control** key while clicking additional objects.

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 7 – Drawing Controls – continued**

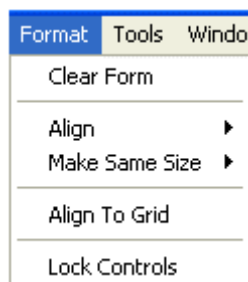
When multiple objects are selected, the primary selected object will appear with blue sizing handles while the additional selected objects will have white handles with blue borders if controls are not locked. When controls are locked, red is substituted for blue.



Selected control functions:

- Performing a second Control-click on a selected object will deselect it.
- Performing a click on a non-primary selected object will designate it as the primary selected object.

Visual PL/B allows you to move, align, size, and delete controls selected as a group once you have selected more than one object. You can use the options available from the Format menu to perform several tasks.

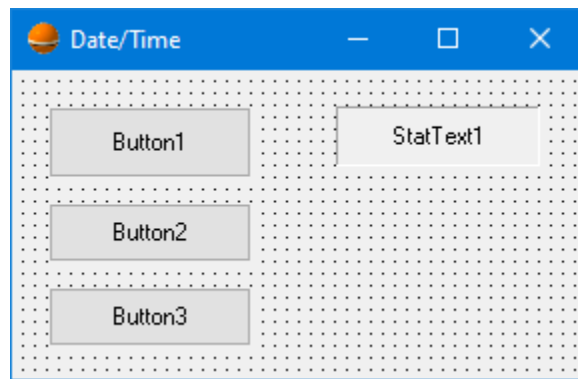




## **EXERCISE – DRAW THE USER INTERFACE**

We will begin with a simple application called Date/Time to demonstrate the use of the IDE:

1. Place three command buttons and one static text box on a blank form.
2. Practice all methods of placing objects on the form.
3. Move and size the controls and the form so that it resembles the form below.

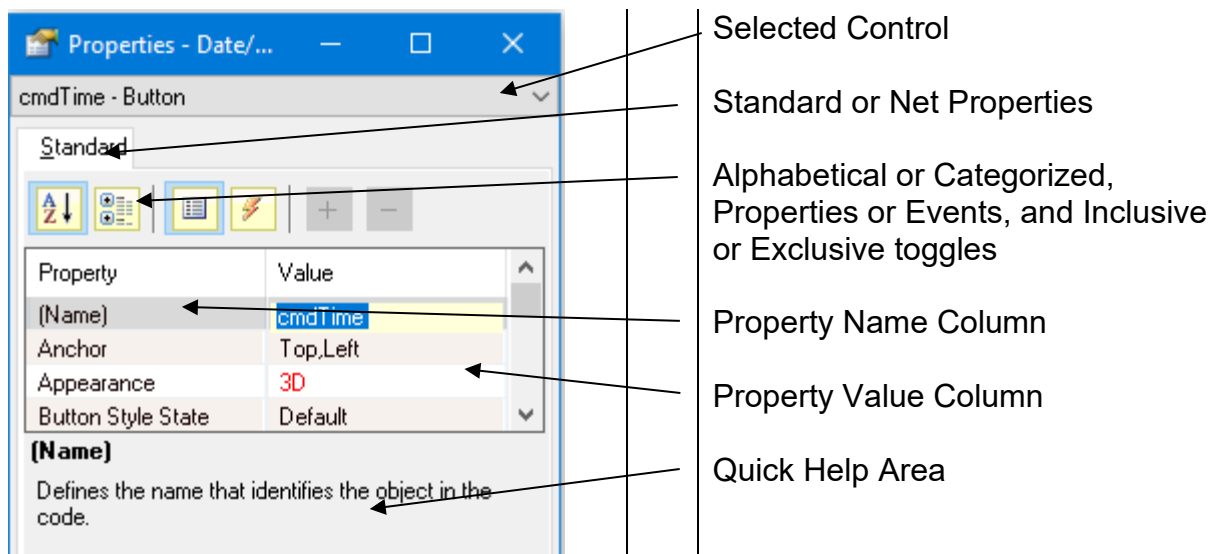


## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 8 – Setting Properties**

Most object properties can be set at design time by changing their value in the settings box of the Properties window. The property window contains several areas of information:

- The combobox at the top shows the selected object and type. If multiple objects are selected “(multiple selections)” is show
- Beneath the combobox is the property type selection tab. When run under PLBNet and when a Net Object control is selected, this tab will allow selection between Standard and Net properties.
- At the top of the tab control are six buttons. From left to right, the first two buttons allow toggling the property list from alphabetical to a categorized format.
- The third and fourth buttons allow toggling between properties and events.
- The fifth and sixth buttons allow toggling between inclusive and exclusive mode when multiple objects are selected. Inclusive mode shows all properties for the objects selected. Exclusive mode shows only the properties the selected objects have in common.
- Modified property values are shown in red or the color selected in the Options dialog.



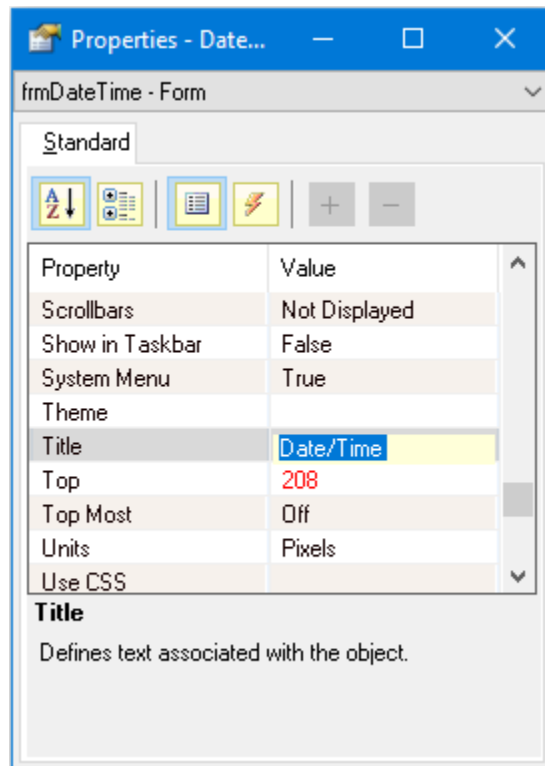
## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 8 – Setting Properties**

There are three basic types of property settings:

1. Text
2. List
3. Dialog

The Title for an object, like a form, is an example of the Text type property. You place your cursor in the settings box and type a new value. For example,

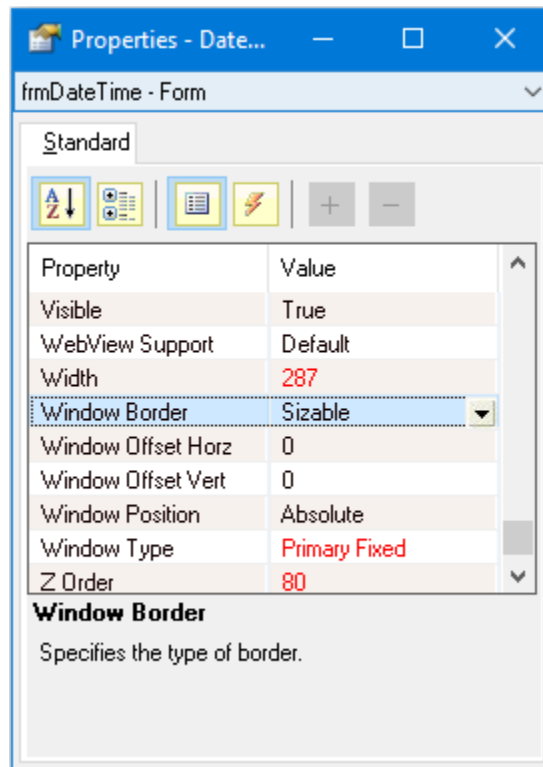


**Changing a text type property**

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 8 – Setting Properties – continued

The Window Border of a form is an example of a list property. Instead of being free form like a text type property, you have a predefined list of choices from which to select. You can drop the list down and select a value or double-click to cycle through the values in the list. Pressing a letter while the dropdown list is visible will position you to the next list item beginning with that letter and select the item.



**List Property**

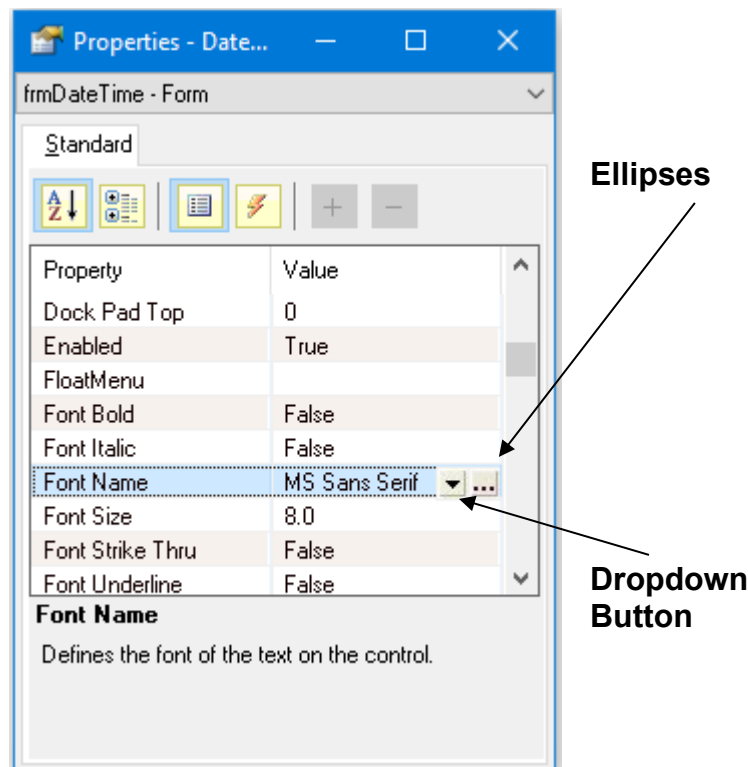
## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 8 – Setting Properties – continued

The Font property is an example of a property that is set using a dialog. When you select the Font property in the Properties window, an ellipse (three dots on a button) appears in the settings box. Click the ellipse to see the Font dialog. Choose the font settings from the dialog and then click OK.

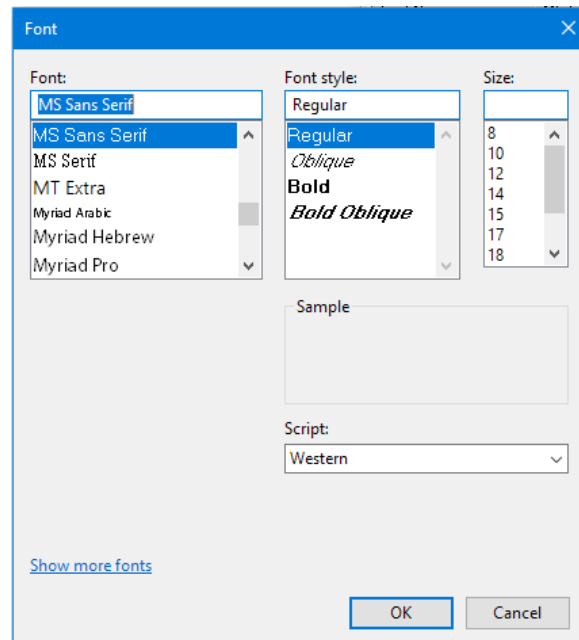
Double clicking the property value will toggle through the available fonts.

Clicking the dropdown button will display a list of fonts from which one may be selected.

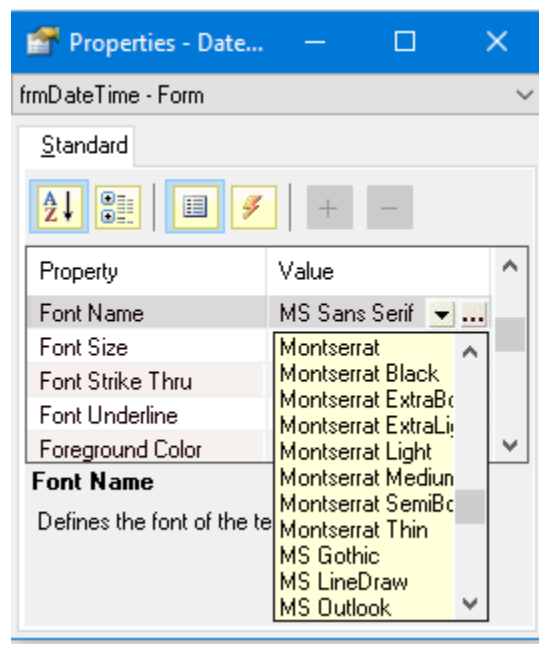


## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 8 – Setting Properties – continued



**Font Dialog**



**Font List**

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 8 – Setting Properties – continued**

You can change properties on each object individually or you can select multiple controls and the Properties window will display all the properties that are common to those objects that you have selected while in Exclusive mode and all properties while in Inclusive mode.

The most important property you can set is an object's **Name**.

The name you assign to an object is how Visual PL/B references that object in your source code. Visual PL/B assigns default names as you create the objects like Form1, Button1, etc. Generally, you will want to change the name property of any object you plan to attach code to or manipulate with code.

Microsoft suggests using a three-character prefix for object name. Some examples for naming objects are:

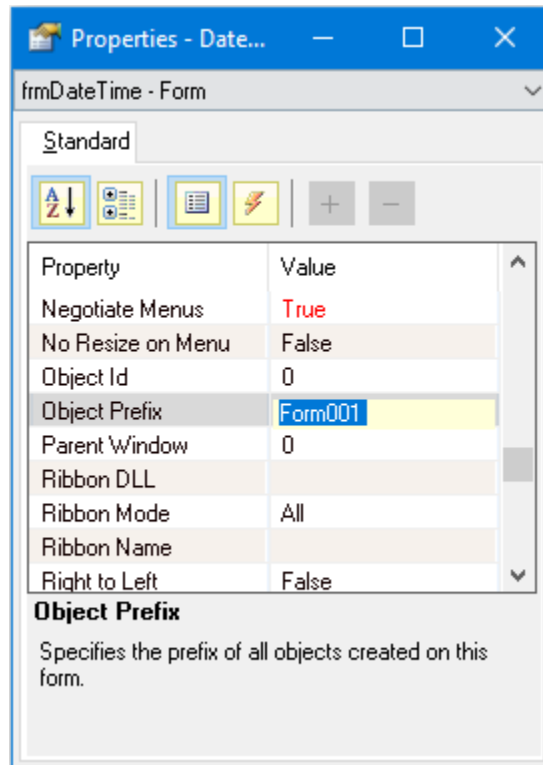
<b><u>Name</u></b>	<b><u>Object</u></b>
frmStopw	Form
txtDollars	Edit Text Box
cmdQuit	Command Button
lblCurr	Static Text
imgDolUp	Image
cboSeat	ComboBox
chkPlayers	CheckBox
lstCity	DataList
rdoRadio	Radio Button
vsbTemp	Vertical Scroll Bar

Object names follow standard PL/B naming conventions. They must begin with a letter and contain only letters, numbers and the underscore (\_) character.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

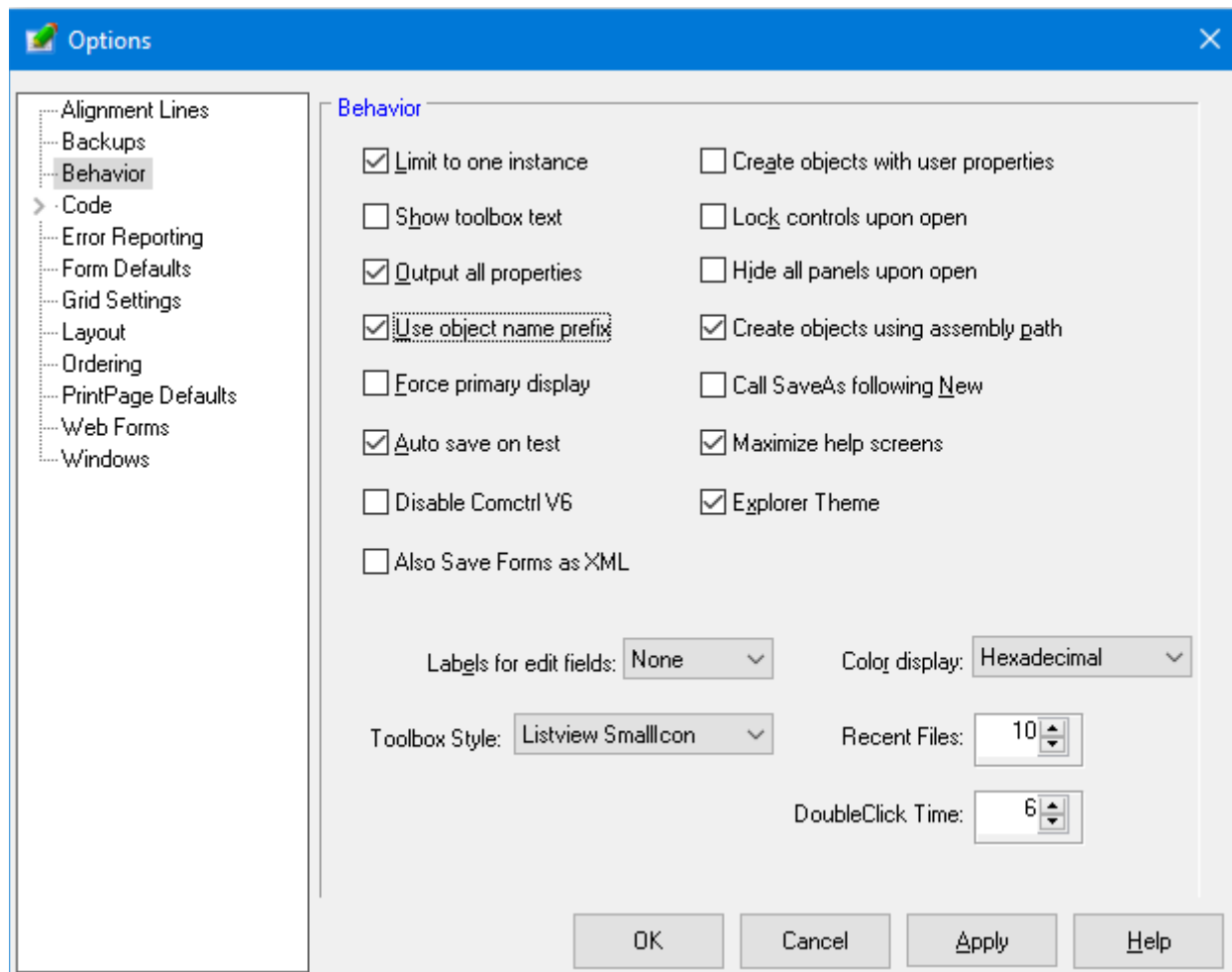
### STEP 8 – Setting Properties – continued

When working with multiple forms within a single program each object must have a unique name. To aid in creating unique names, each form has an Object Prefix property



To employ the Object Prefix property, both the value must be defined and the option to use the property set on the options behavior tab. Once enabled, this property's value is pre-pended to any object as it is created.





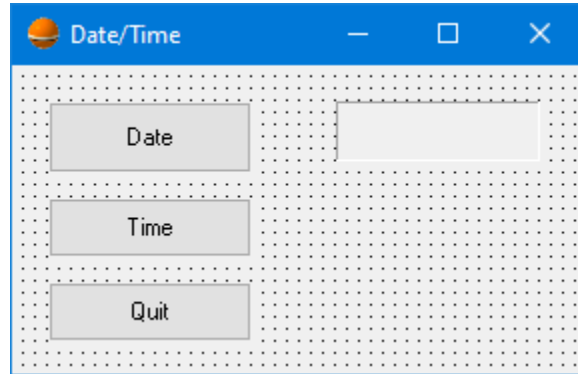
**EXERCISE – Setting Properties**

The purpose of this exercise is to set the properties of the form and objects in our Date and Time interface. Use the list below as your guide for setting properties for the applications. As you change the names of the objects, their new names will appear in the object box of the properties window.

<b><u>Object</u></b>	<b><u>Property</u></b>	<b><u>New Setting Value</u></b>
Form1	Name	frmDateTime
	WindowType	Primary Fixed
	WindowBorder	Fixed Single
	Title	Date and Time
StatText1	Name	lblDisplay
	Alignment	Center
	Border	True
	Style	3D On
	Text	(erase everything)
Button1	Name	cmdDate
	Title	Date
	ToolTipText	Display the date.
Button2	Name	cmdTime
	Title	Time
	ToolTipText	Display the time.
Button3	Name	cmdQuit
	Title	Quit
	ToolTipText	End the program.

**EXERCISE 3 – Setting Properties – continued**

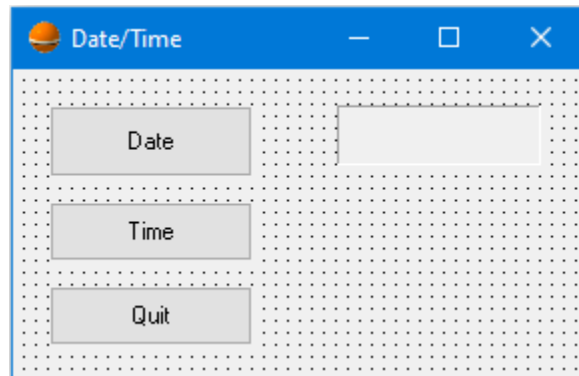
Once complete, your form should appear as:



## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 9 – Writing Code for Your Application**

The next step in our application development process is to make our application respond to the user actions.



When the user clicks the Date command button, we want to display the date in the StatText object we named lblDisplay.

We have to add code using the code window to the Date, Time, and Quit command buttons in order to make them respond to the user.

The quickest way to open the Code window is to double-click on the object for which you wish to write code.

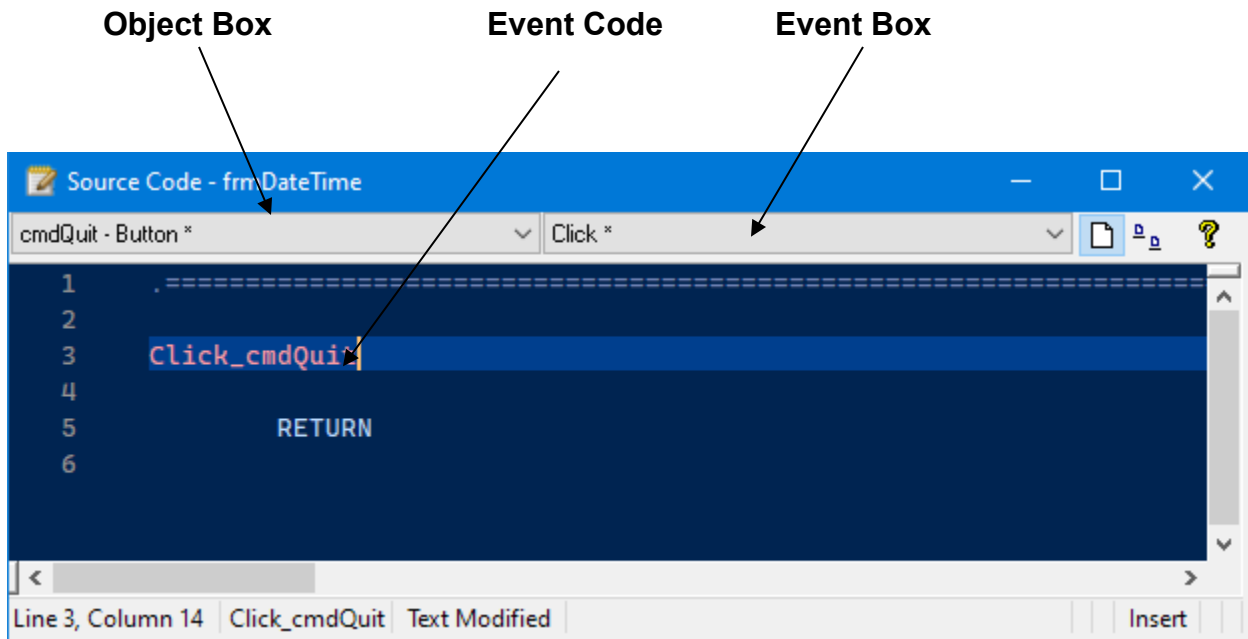
#### **NOTE**

To prevent accidental movement, you may want to Lock Controls at this point.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 9 – Writing Code for Your Application - continued

Double-clicking the Quit button will produce a code window as follows:



**Code Window**

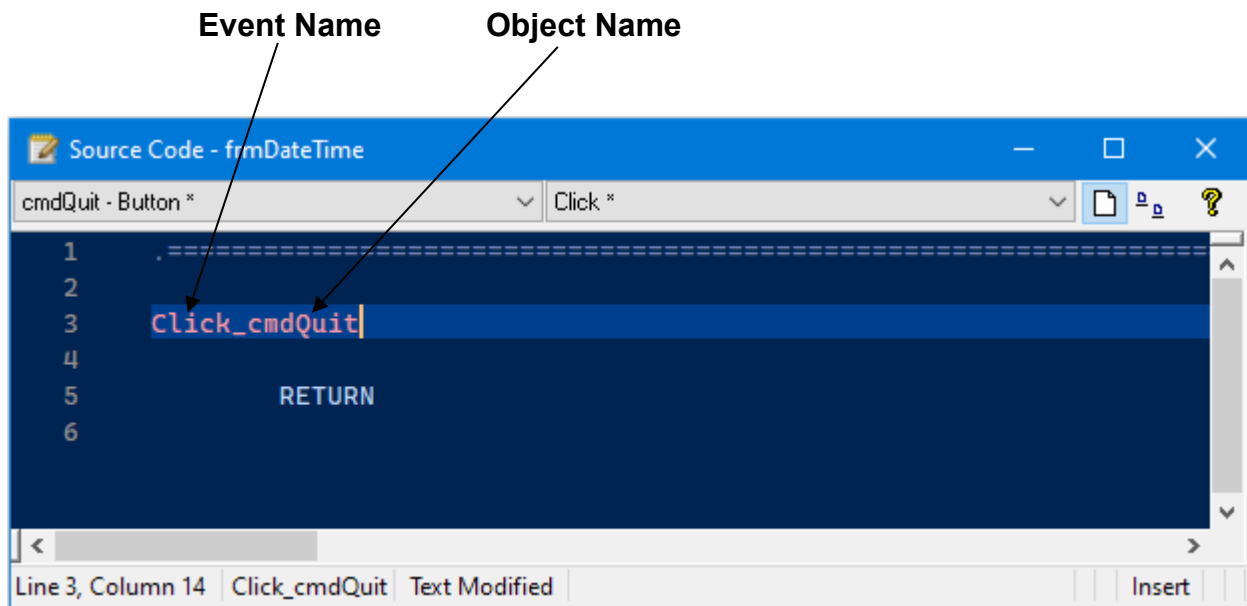
Code is added to objects as event routines. We will talk about different events in a subsequent chapter, but the default event for a command button is the Click event. Visual PL/B opened the Code window to that event for us to begin writing code.

This click event routine is code that will be executed when the user clicks on the Quit command button.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 9 – Writing Code for Your Application - continued

Event routine names are constructed by Visual PL/B to include the object's name and the name of the event to which the routine is attached.

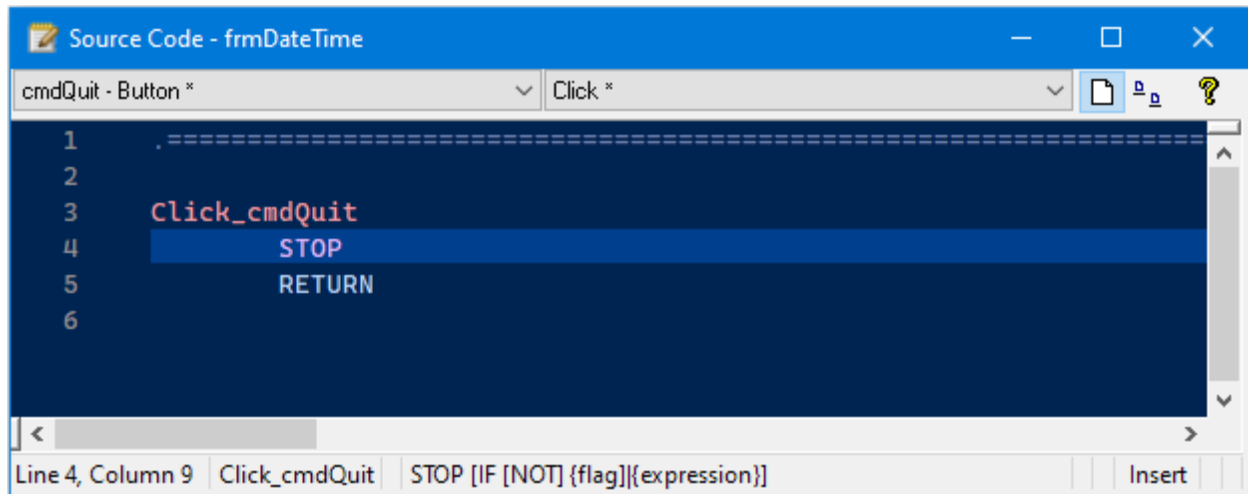


Your job is to write one or more code statements that tell cmdQuit how to respond when it is clicked. The cursor is blinking between the routine label and the RETURN statement waiting for you to write code.

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 9 – Writing Code for Your Application - continued

When you click the Quit button, you want to end the application. To accomplish this, we simply insert a STOP statement into the routine.



```
Source Code - frmDateTime
cmdQuit - Button * Click *
1  .=====
2
3  Click_cmdQuit
4      STOP
5      RETURN
6
```

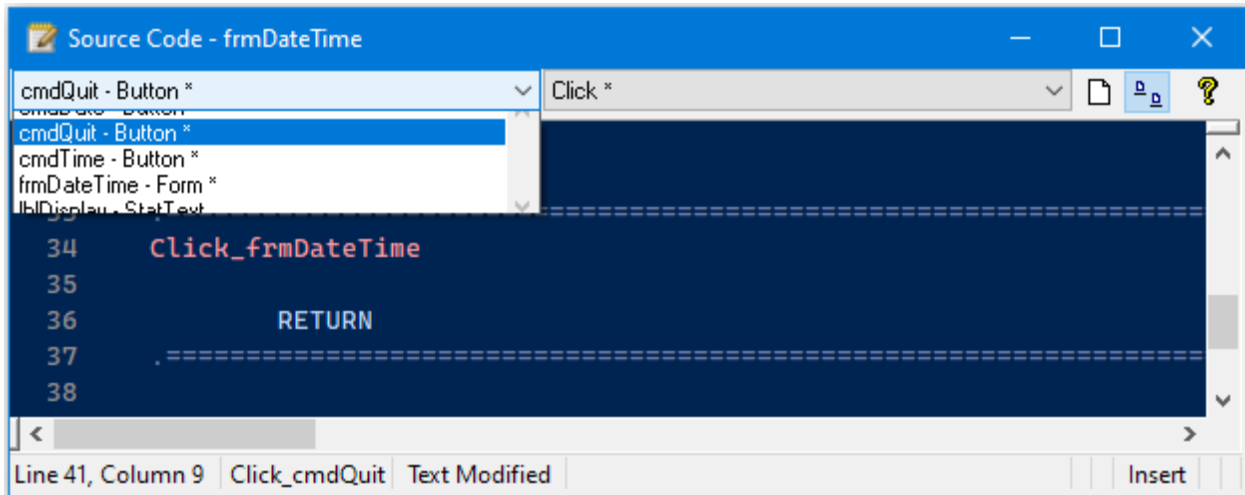
Line 4, Column 9 | Click\_cmdQuit | STOP [IF [NOT] {flag}]{expression} | Insert

#### Modified Event Routine

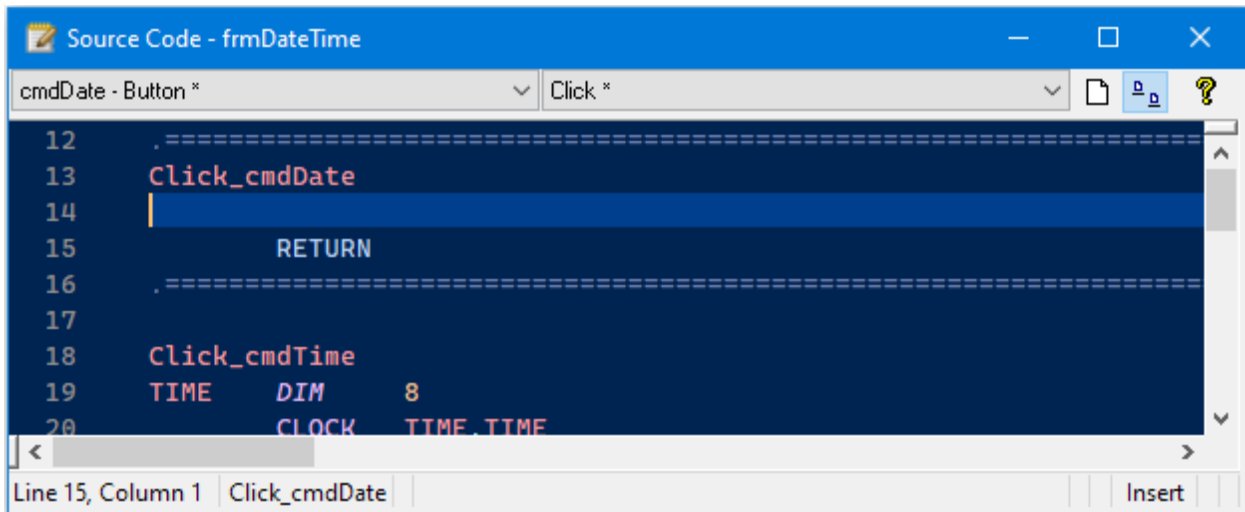
## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 9 – Writing Code for Your Application - continued

To write code for the other objects, you can click the object box in the Code window and select the other objects one at a time. You may then add code to their event routines.



### Selecting another object

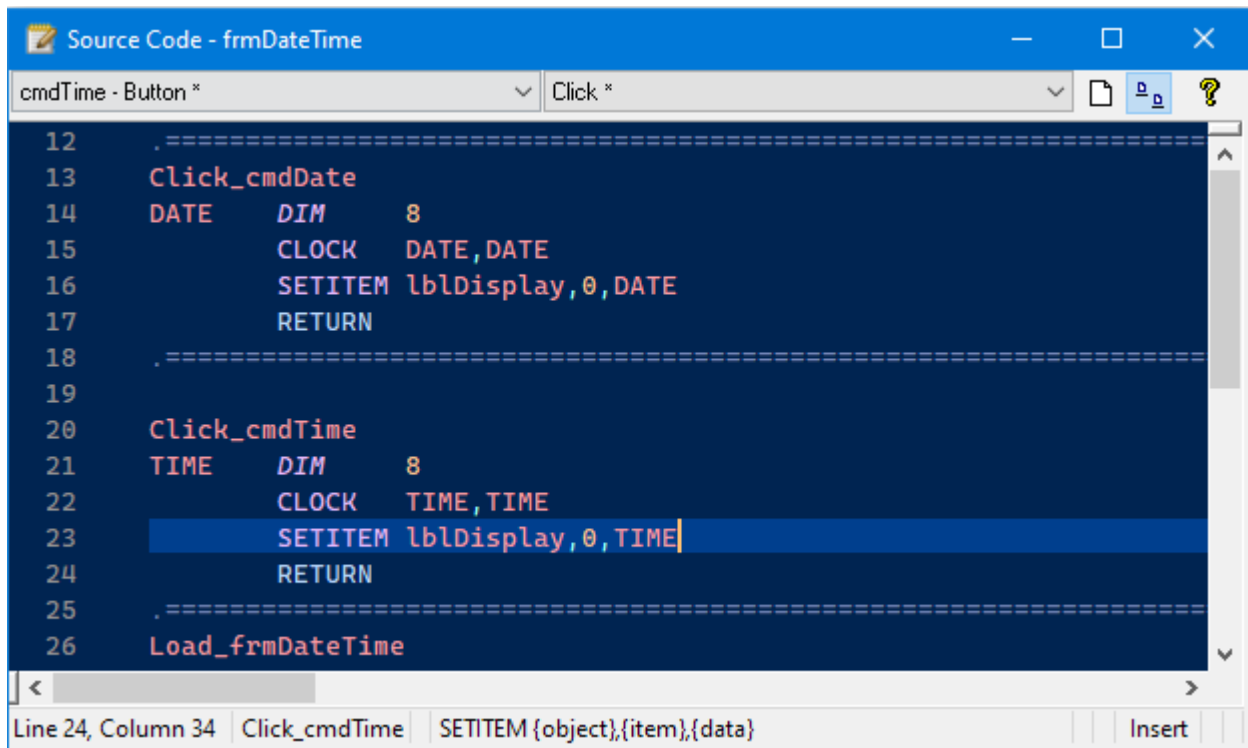




## EXERCISE – WRITING CODE

You need to write code to display the date and time. The code will need to create a variable and then retrieve the appropriate value using the CLOCK instruction. Once the value has been obtained from the system, we need to place it into the StatText object named "lblDisplay" using the SETITEM instruction.

For the date and time command buttons, write the following code exactly as it appears in the click event to display the current machine time and date in the label.



```
Source Code - frmDateTime
cmdTime - Button * Click *
12 .=====
13 Click_cmdDate
14 DATE DIM 8
15 CLOCK DATE,DATE
16 SETITEM lblDisplay,0,DATE
17 RETURN
18 .=====
19
20 Click_cmdTime
21 TIME DIM 8
22 CLOCK TIME,TIME
23 SETITEM lblDisplay,0,TIME
24 RETURN
25 .=====
26 Load_frmDateTime

Line 24, Column 34 Click_cmdTime SETITEM {object},{item},{data} Insert
```

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

### **STEP 10 – Saving Your Form**

The quickest way to save the current form is to click the Save button on the Form Designer's toolbar. Remember to save your work often.

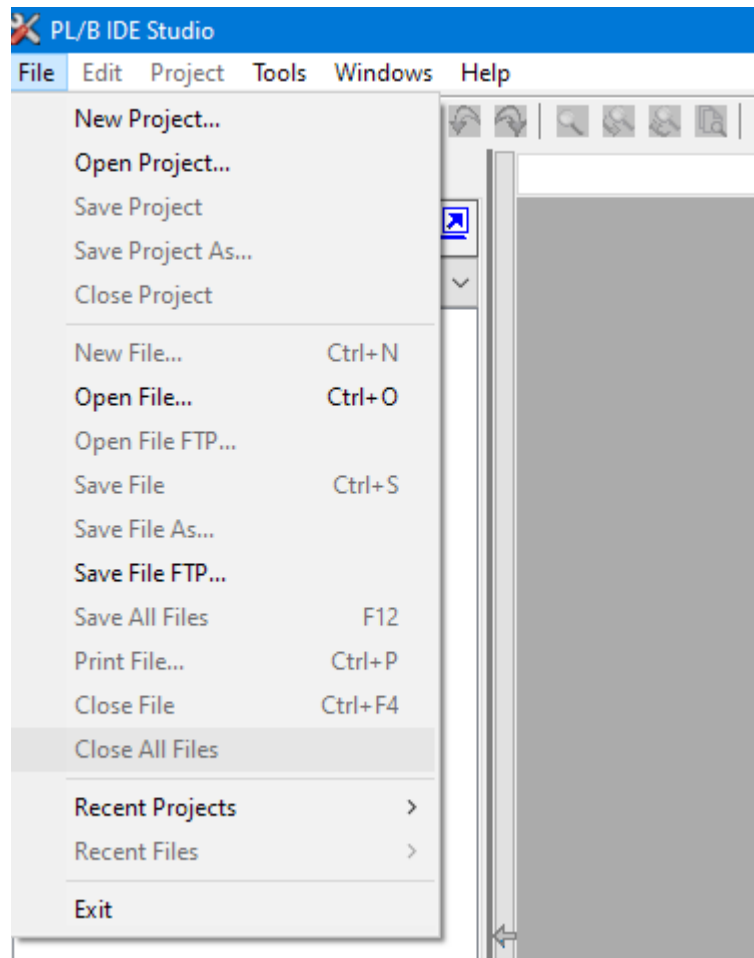


#### **Save Form**

The form's name and directory are displayed in the Title Bar of the Designer's main window. Ensure that the values are correct before clicking Save.

## **EXERCISE – Saving Date/Time**

1. If you have not done so already, save the Date/Time form and project files.
2. Close Visual PL/B.
3. Start Visual PL/B and choose your Date/Time project from the list of recent projects.

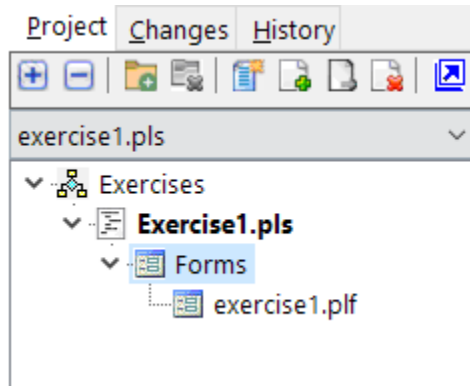


4. When your project opens, you will need to open your form again by double-clicking it in the Project window.
5. A prompt may appear asking if you want to create a backup directory. You should respond by clicking "Yes".

## **VISUAL PL/B APPLICATION DESIGN STEPS – continued**

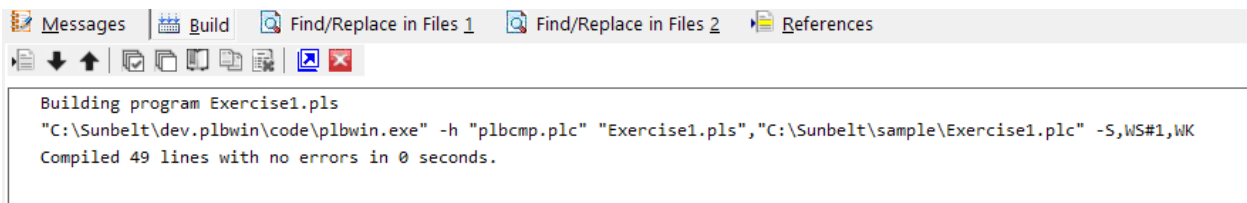
### **STEP 11 – Building Your Application**

To compile your application, first make sure "exercise1.pls" is selected in the Project window. To select the source file, simply click it. The name will then become highlighted to show it as selected. The form will appear after the source file is built for the first time.



Once the source file is selected, choose Build Exercise1.pls from the Project menu or press the F7 key.

The build process invokes the PL/B compiler. Progress messages and errors are reported in the message window at the bottom of the IDE. You may need to enlarge the window or scroll it to see all the error messages reported. Look for the “no errors” message in the window.



### **Message Window**

## VISUAL PL/B APPLICATION DESIGN STEPS – continued

### STEP 12 – Running Your Application

Once your application compiles with no errors, you may run the program by choosing Execute from the Project menu or pressing F5.

Steps 11 and 12 are commonly performed using the Build button and the Execute button on the Toolbar. These buttons will recompile any out-of-date source file and execute the application.



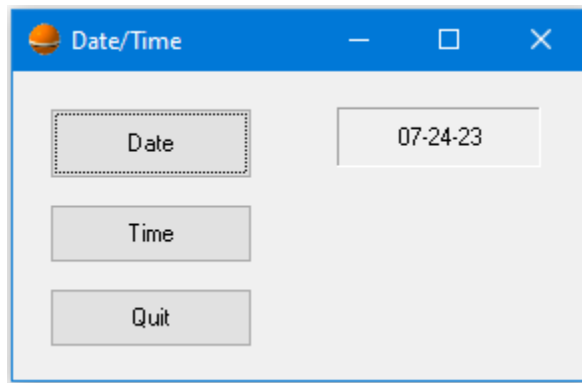
You may notice a second window that appears when your program executes. This window is called the Main Window. Any DISPLAY statements encountered will output to this window. Generally, the window is not required or desired when running GUI programs such as our first exercise. The Main Window may be hidden by adding a WINHIDE instruction to the program source file.

```
MAIN PLFORM      EXERCISE1.PLF
.
  WINHIDE
  FORMLOAD      MAIN
.
  LOOP
  EVENTWAIT
  REPEAT
```

### Modified Source Code

**EXERCISE – BUILDING AND RUNNING DATE/TIME**

1. If you have not done so already, build your Date/Time application.
2. Once the program compiles correctly, test the program by running it.



**CHAPTER FOUR**

***FORMS, CONTROLS, AND MENUS***

## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter describes the different types of objects you can use in designing your Visual PL/B Application.

Upon completion of this chapter, you will be able to:

- Identify the various forms, controls, and their properties.
- Use some of the controls in simple projects.
- Add ActiveX and .Net controls to a project.
- Understand how to create a menu for a form.
- Demonstrate how to add, remove, and display forms.
- Learn how to access the PL/B Language Reference.



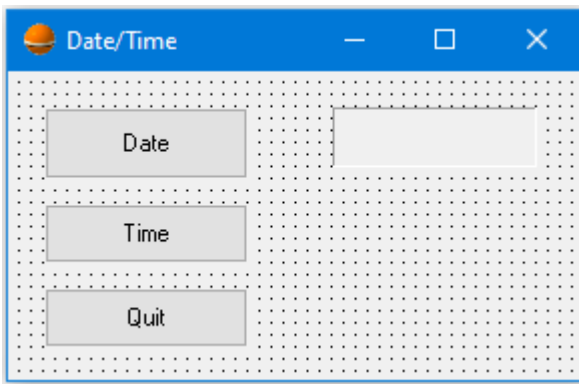
## VISUAL PL/B FORMS

Forms or Windows that you design are the basis for your Visual PL/B application.

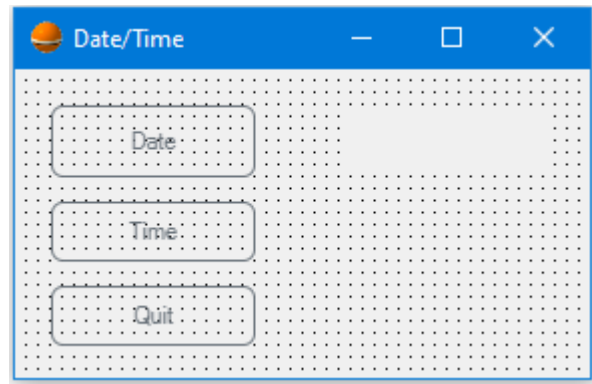
Forms and controls display information, gather information from the user, and process information in your application.

Forms have properties that can be set at design time or run time.

There are two types of forms: standard and web. We will work with standard forms in this class.



**Standard Form (.plf)**



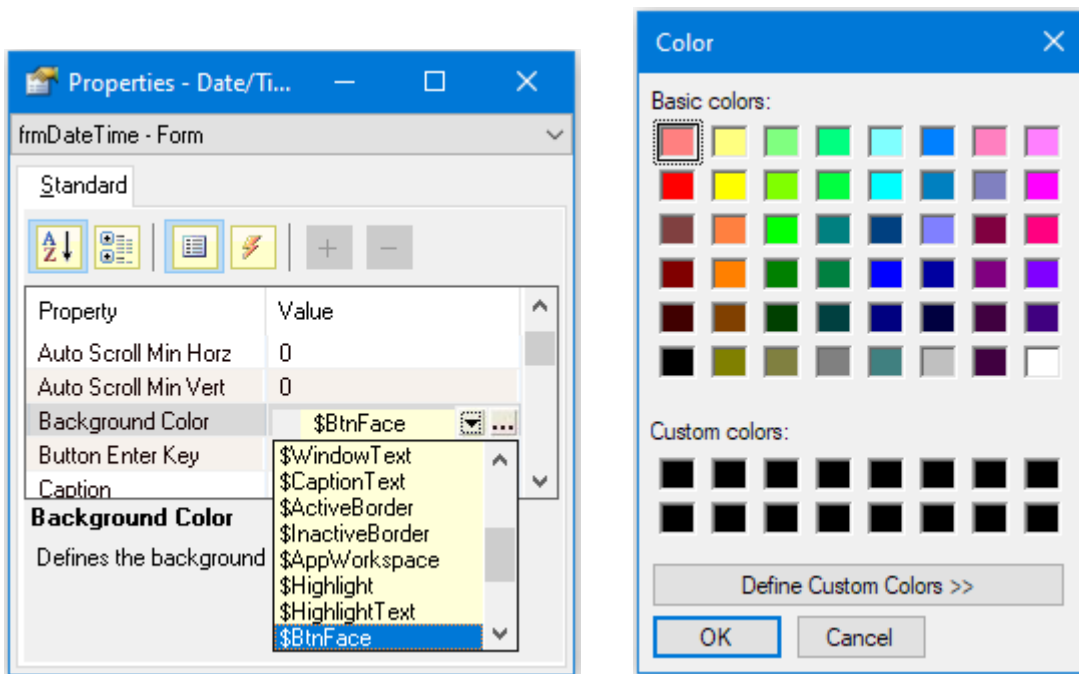
**Web Form (.pwf)**

## VISUAL PL/B FORMS - continued

The form on which you created your user interface has many properties that control the appearance and behavior of the form. Some properties can be set at design time only. Others can be set at run time. All of the properties that you see in the Properties window are design time properties. Some of the design time properties for forms are listed below:

**Appearance** – 3D effect or not

**BackColor** – Set the background color of a form using the color list or color palette.



**WindowState** – Set the window to be fixed or sizeable and allow the program to appear on the taskbar or not. While this property may be set at design time, the effect is not visible until run time.

**WindowBorder** – Set the window border to be fixed or sizeable. While this property may be set at design time, the effect is not visible until run time.

## **VISUAL PL/B FORMS - continued**

**Title** – Text displayed in the form's title bar.

**Enabled** – Warning – If you set Enabled to False neither the form nor any of the objects on the form will respond to events. This is probably not a desired effect.

**Height, Width** – Measures the height and width of the form in units based on the form's Unit property.

**Maximize Box, Minimize Box** – Set to True if you want these buttons to be present on the form. The WindowType property will override these settings. If you choose a fixed WindowType property, you must specifically set these boxes to true if you want to minimize or maximize the form. The effects of these properties are not visible until execution time.

**Name** – The name used in all code statements to refer to the form object. This property may not be changed at run time.

**Window Position** – Specifies where you want the form to appear when it becomes visible.

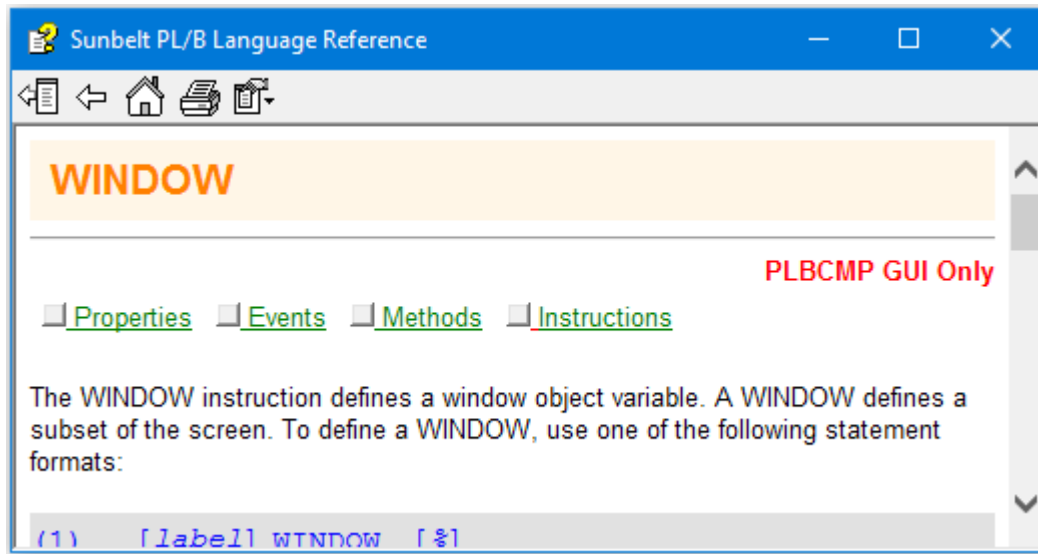
**Top, Left** – Location of the form on the Windows screen in twips.

Many objects like forms have run time properties. An example of a run time property for a form is the Hwnd property that contains the Windows handle of the form.

**Font** – Defines the default font for all objects placed on the form.

## VISUAL PL/B FORMS - continued

You can always use the PL/B Language Reference to look up the properties, events, and methods for any object or to understand the effects of property settings.



### PL/B Language Reference

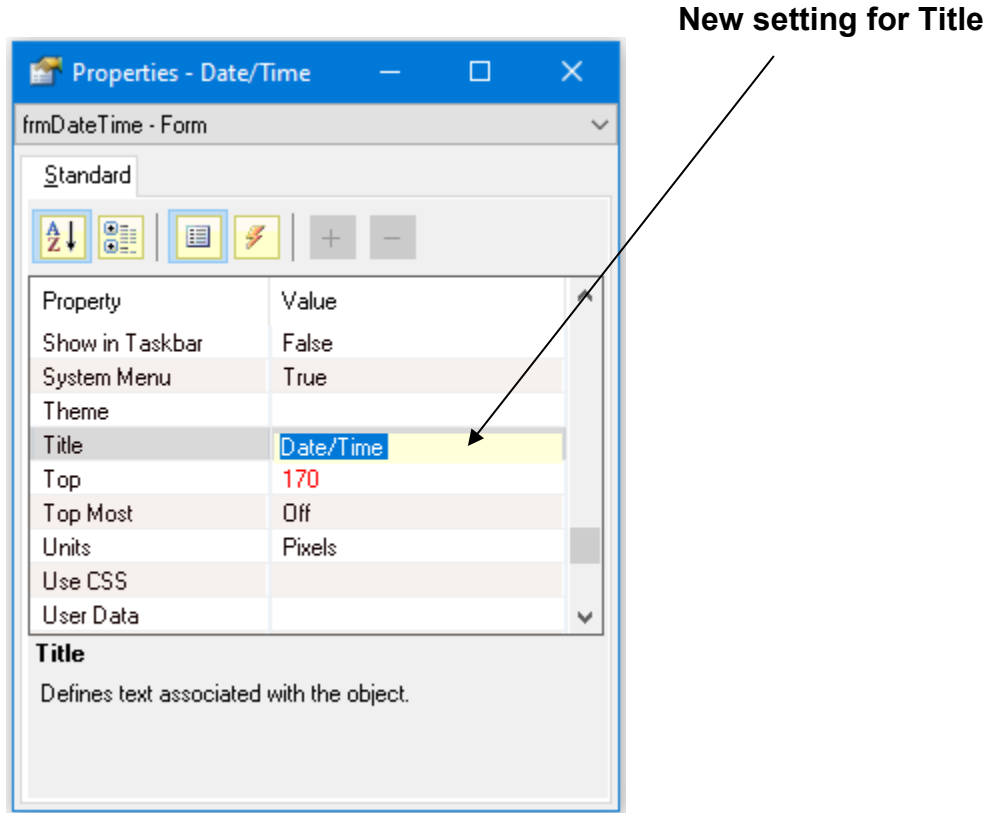
To display reference information regarding an object, click the object in the Toolbox and then press F1.

To display information regarding a property, click the property in the Property window and press F1.

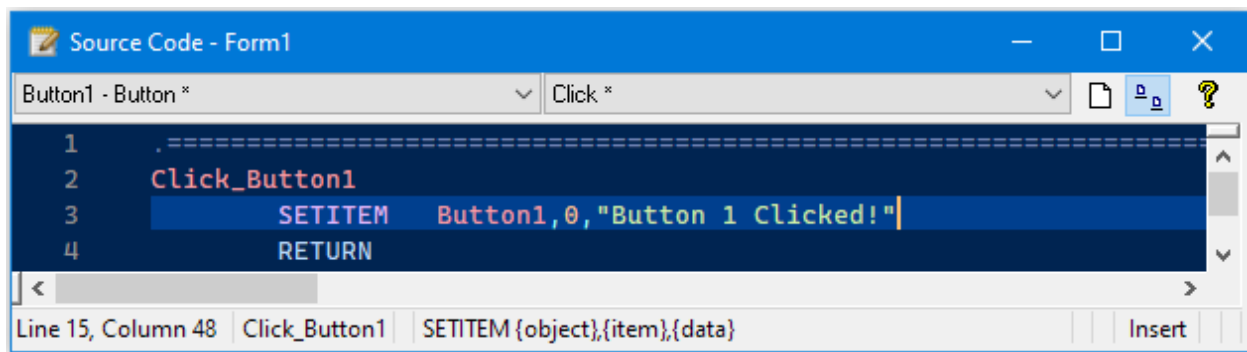
PL/B Language help is also available in the code window of both the IDE and the Form Designer. Simply highlight an instruction and press the F1 key.

## SETTING PROPERTIES

To set properties for an object at **design time**, you change the selected property's value in the Properties window.



To set properties at **execution time**, you must use a **SETITEM** or **SETPROP** statement in your code. These two instructions work together to expose all the properties of an object. In general, SETITEM changes the value of an object and SETPROP changes the attributes (colors, border, etc.). For example, if you wanted to change a command button's title at run time when you click the button, you would code the following statement in the click event:



**Example of a run time property change**

## **STANDARD CONTROLS**

The standard controls in the Toolbox are built into Visual PL/B and always available for use. The Standard Set of Controls include:

### **Pointer**

This is the only item in the toolbox that doesn't draw a control. When you select the pointer, you can only resize or move a control that has already been drawn on the form.

### **Button**

Creates a button the user can press to initiate a process.

### **Checkbox**

Creates a box that the user can choose to indicate a selected item.

### **Radio**

Allows you to display multiple choices from which the user can choose only one.

### **EditText**

Holds text that a user can enter or change.

### **StatText**

Allows you to have text that you don't want the user to change such as a caption under a graphic

### **DataList**

Displays a list of items from which the user can choose. The list can be scrolled if it has more items than can be displayed at one time.

## **STANDARD CONTROLS - continued**



### **ComboBox**

Allows you to draw a combination DataList and EditText. The user can either choose an item from the DataList or enter a new value in the EditText section.



### **VScrollBar**

Provides a graphical tool for quickly navigating through a long list of items.



### **HScrollBar**

Provides a graphical tool for quickly navigating through a long list of items.



### **Pict**

Displays a graphical image on a form.



### **Icon**

Displays a graphical image on a form.



### **GroupBox**

Allows you to create a graphical object for the functional grouping for controls.



### **Progress**

Allows you to graphically represent the progress of a specific process



### **Shape**

Allows you to draw a variety of shapes on your form at design time. You can choose a rectangle, rounded rectangle, square, rounded square, oval, or circle

## **STANDARD CONTROLS - continued**

### **Line**

Used to draw a variety of line styles on your form at design time.

### **TabControl**

Allows arranging objects on a number of tabbed pages that are selected one at a time by the user.

### **Slider**

Allows user selection via a horizontal or vertical slide bar.

### **MRegion**

Allows the capture of mouse events within a specified region of a form.

### **ListView**

Arranges data in a tabular list form.

### **TreeView**

Provides a tree structure for the display of hierarchical data such as files and directories.

### **OLE Container**

Allows you to link and embed objects from other applications into your Visual PL/B application.

### **StatusBar**

Provides an area for status messages and other information that is relayed to the user.



## **STANDARD CONTROLS - continued**



### **ToolBar**

Allows the designation of icons to provide shortcuts to program functions.



### **Splitter**

Provides a mechanism to adjust the widths or heights of adjacent panels.



### **Panel**

Creates a palette onto which other objects may be placed and dealt with as a group.



### **EditNumber**

Provides an input device for several types of numeric values.



### **EditDateTime**

Allows input of date via the keyboard by selecting from a calendar



### **LabelText**

Defines text that is for display only and can't be changed by the user.



### **Animate**

Provides an animated graphic on the form.



### **Timer**

Creates a non-displayed object that will trigger an event at a given time interval.



### **ImageList**

Groups a number of images together as a unit that may then be associated with another object such as a ToolBar

## **STANDARD CONTROLS - continued**



### **FloatMenu**

Creates a menu that may appear anywhere within the form.



### **Menu**

Defines a menu that appears at the top of the form.



### **SubMenu**

Creates subordinate menus for Menus and FloatMenus



### **RichEditText**

Creates an edit text that allows enhanced text formatting.



### **HTMLControl**

Creates a container which can be populated with HTML elements.



### **DataTable**

Creates an object that provides a means of loading, storing, and manipulating a table of data that is maintained and referenced using a row and column orientation.

## **STANDARD CONTROLS - continued**

### **Command Buttons**

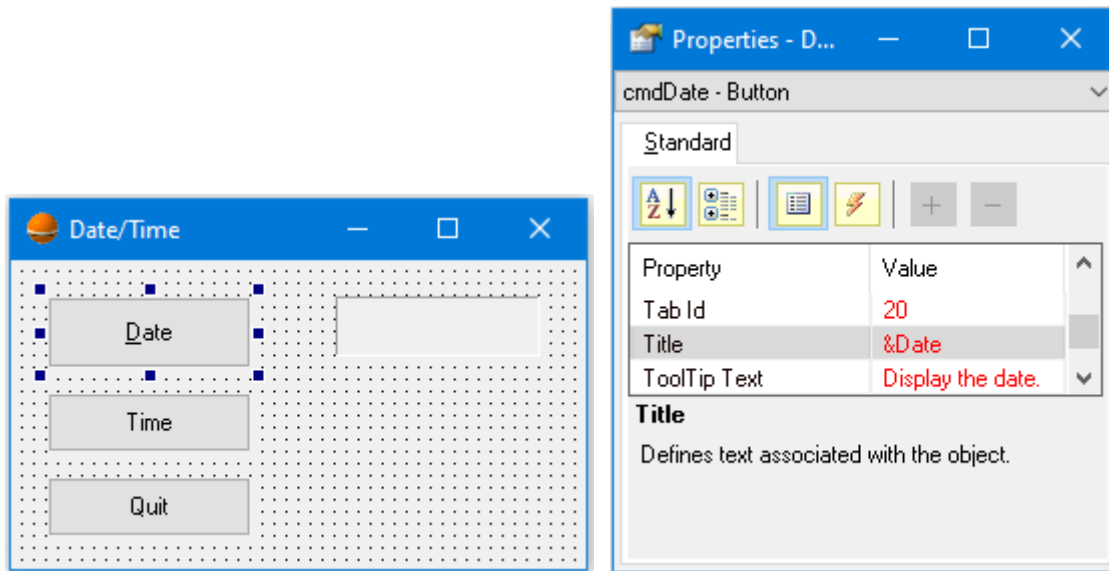


Many Visual PL/B applications have command buttons that allow the user to click them to perform actions. When the user chooses the button, it not only carries out the appropriate action in the Click event procedure, but it also looks as if it is being pushed in and released. It is sometimes referred to as a push button.

Buttons can contain text or a graphic. Text buttons usually appear in the client area of the interface and graphic buttons are mostly used in toolbars.

In normal usage, the user points to and clicks once on a command button and Visual PL/B then executes the code in the command button's click event procedure.

Access keys can also be assigned to the text of a command button. By coding an ampersand (&) before the desired letter in the text, you enable the mnemonic feature of the control. The sample below shows three text buttons. The click event for the Date button can be activated by the user pressing ALT+D because that is access key assigned to the button.



**Access Key Definition**

## STANDARD CONTROLS - continued

### Command Buttons - continued

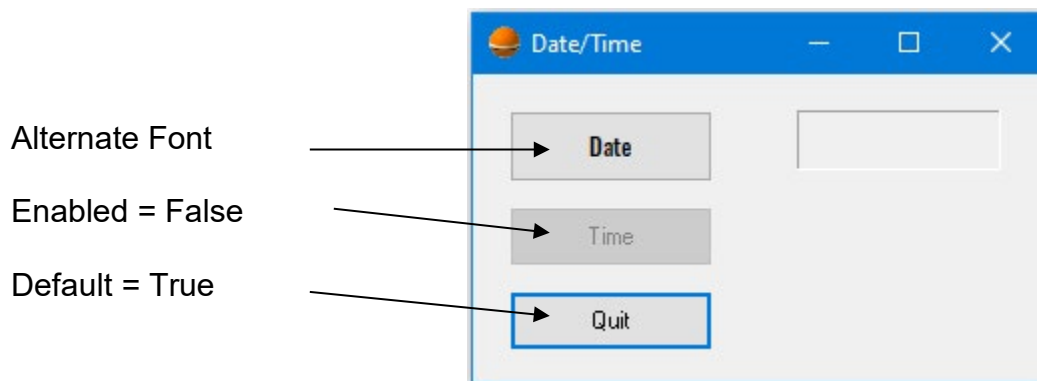
**Anchor** – The Anchor property ties an object to one or more edges of its parent object.

**Default** – On each form, you can designate one command button as the default command button. That is, if the user presses the ENTER key on the keyboard, the command button is clicked regardless of which other control on the form has the focus. To specify a command button as the default, set the Default property to True. The default button on a form appears to the user with a dark border.

**Cancel** – You can also specify one button on a form as the cancel button. When the cancel key (escape) is pressed, a click event is generated for the designated cancel button regardless of which other control has the focus. The specified cancel key has no visible attributes at execution time.

**Enabled** – When the Enabled property is False, the command button will not respond to user actions. The sample below shows the Time button's appearance (faded at execution time) when the Enabled property is False.

The **Font** properties for many objects like command buttons allows specification of alternate fonts, font sizes, and effects for the text that appears on the object.



## STANDARD CONTROLS - continued

### StatText and LabelText

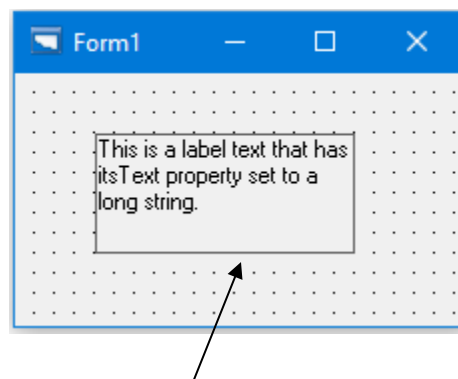
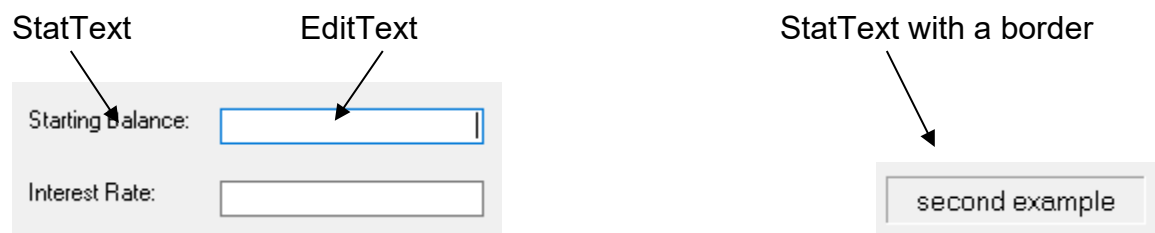
**A**

StatText and LabelText objects are often used as a method of identifying another control on the form. They are sometimes generally referred to as labels. Both objects provide the same basic functionality with LabelText having a larger set of properties.

StatText objects do not support user entry so they can also be used when you want to display text to the user but do not want the user changing the value.

StatText objects do not support user selection although it is possible to respond to a mouse event such as a click.

Below are some examples of StatText objects;



**A LabelText object with its WordWrap property set True**

## **STANDARD CONTROLS - continued**

### **StatText and LabelText Properties**

**Alignment** – This property determines how the objects text property is positioned:

- Left justified.
- Right justified.
- Centered

**BackStyle** – Transparent allows the background color of the form to show.

**UseMnemonic** – When this property is True, an access key designation (&) is allowed in the Text property. At run time, pressing ALT+ the access key triggers the click event for the STATTEXT object. You can then use the SETFOCUS instruction to move the focus to the appropriate EDITTEXT object.

**BackColor** – Defines the background color used when the BackStyle property is set to Opaque.

**ForeColor** – Define the color of the text displayed in the object.

**EXERCISE – SETTING PROPERTIES**

1. Open the Date and Time form from the previous chapter.
2. Set the properties to make the Date command button the Default button and the Quit command button the Cancel button.
3. Add code to the Click event procedure for the Date button that will set the form's title to "The current date is:". Do not remove your existing code.
4. Add code to the Click event procedure for the Time button that will set the form's title to "The current time is:". Do not remove your existing code.
5. Assign access keys to the titles of the Date and Time buttons.
6. Set the Alignment, Font, and ForeColor properties to any values you wish for the StatText object.
7. Save your form.
8. Test your project.
9. This completes the exercise.

## STANDARD CONTROLS - continued

### CheckBoxes, Radio Buttons and Groupboxes



A CheckBox allows the user to turn on or off some value. These objects act as a toggle switch. If more than one CheckBox appears on the interface, the user can choose to have one, all, or none of the checkboxes selected.

Use a CheckBox when you have a single piece of information that has two options. For example, in the interface below, CheckBoxes allow the user to specify whether they want the text to be bold and/or underlined.

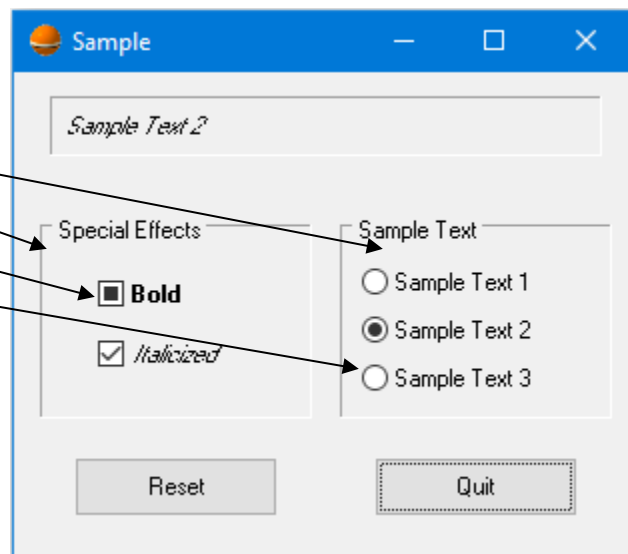
Radio Buttons (sometimes referred to as option buttons) also allow the user to make a choice among several values. Radio Buttons are normally used in a group of two or more in which only one Radio Button at a time is selectable (like a car radio). Selecting one Radio Button from a group automatically de-selects any previously set button in the group. Thus, only one button at a time may be set.

Use Radio Buttons when the user among several mutually exclusive options. For example, in the interface below, only one sample text can be chosen to be display at a time. GroupBoxes are normally used to isolate the options in the group from other form controls.

GroupBoxes

CheckBoxes

Radio Buttons





## **STANDARD CONTROLS - continued**

### **CheckBoxes, Radio Buttons and Groupboxes - continued**

Radio Buttons and CheckBoxes have Title, BackColor, ForeColor, Font, and Enabled properties that can be set to affect their appearance and behavior.

The Value property for Radio Buttons and CheckBoxes determines their state. For Radio Buttons, their state is either True or False. True means it is selected. Only one Radio Button in a group may be set to True at any given time.

#### **NOTE**

For Radio Buttons to work as a group, the GroupID property must be set to the same value for each button.

Checkboxes may have three possible value property settings if the TriState property is set to True.

0 – Unchecked

1 – Checked

2 – Grayed Check

## STANDARD CONTROLS - continued

### EditText, EditNumber, and EditDateTime



EditText, EditNumber, and EditDateTime objects are useful for retrieving information from your user because unlike StatText objects they support user input.

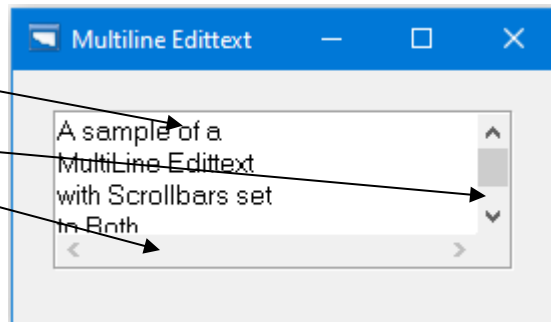
EditText objects can be either single line or multi-line. To make an EditText support multiple lines, set the MaxLines property to the number of lines desired. The default value is one (1) line.

Once you have set MaxLines to value greater than one, you can also set the Scrollbars property to provide a vertical scrollbar, a horizontal scrollbar, or both.

EditText objects do not have a title property. Instead, they use a Text property to indicate the contents of the EditText object. StatText or LabelText are often employed to inform the user as to the expected content of the EditText object.

MultiLine EditText

ScrollBars = Both



## **STANDARD CONTROLS - continued**

### **EditText, EditNumber, and EditDateTime – continued**

Other properties that affect the appearance and behavior of EditText objects are listed below.

**ReadOnly** – When set to True, disallows user input only. Users can scroll, highlight, and copy but cannot cut, paste, or edit in the control.

**Static** – When set to True, disallows user input. Users can scroll but cannot copy, cut, paste, or edit in the control. Additionally, the control cannot receive focus.

**Maximum Characters** – Usually set to zero (0) which allows 2048 characters unless the MaxLines property is set in which the limit is about 64K characters. Setting the Maximum Characters to anything other than zero limits the length of the text field. No error will occur if the user types too many characters. The extra characters are simply truncated. Changing the property at run time will not affect the current contents but will affect any subsequent changes.

**Word Wrap** – Setting this property to True allows the automatic wrapping of long lines when the MultiLine property is greater than one (1).

There are other properties, events, and methods of EditText objects that we will explore in a subsequent chapter.

## **STANDARD CONTROLS - continued**

### **EditText, EditNumber, and EditDateTime – continued**

EditNumber objects are designed for numeric input. Beyond input filtering for valid values and support of the basic EditText object attributes, several properties enhance these objects.

**Integer Digits, Decimal Digits** – These properties allow control of the number format.

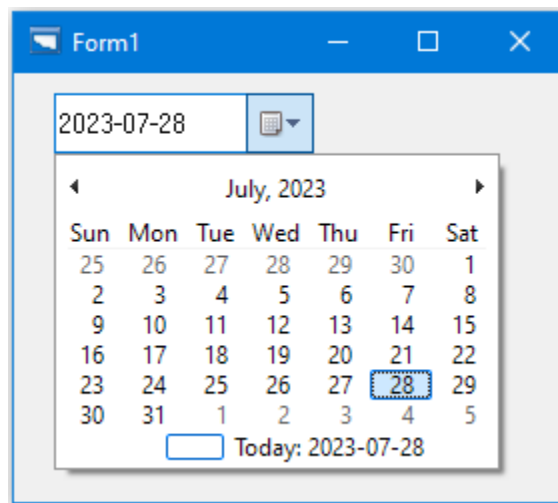
**UpDownAlign, UpDownMin, UpDownMax, UpDownChange** – These properties work in unison to provide an up/down control that is associated with the EditNumber. The UpDownMin and UpDownMax properties establish limits on the control while the UpDownChange controls the increment or decrement value. Before the up/down control is visible, its UpDownAlign property must be set to something other than None.

An EditDateTime object is a variation of an EditText designed to aid the user with the input of a date or time. Beyond simple date validation, the control provides several features.

**Minimum Date, Maximum Date** – These properties ensure the date input will be within a specified range.

**Custom Format** – This property defines a format string for the data displayed in the object.

**Cal FG Color, Cal Month BG Color, Cal Title BG Color, Cal Title FG Color, and Cal Trailing Color** allow custom color definitions for the dropdown calendar.



## **STANDARD CONTROLS - continued**

### **Icons and Picts**



Icon and Pict objects are often used to place graphics in a specific location on the form.

You can load a picture into the object by setting the **ResourceID** property. The ResourceID is defined when the image is added to the form's resource via the Resources item of the Tools menu.

Icons support only icon (.ico) files. Picts support icons (.ico), run-length encoded files (.rle), bitmaps (.bmp), and JPEG (.jpg) files.

Visual PL/B has a number of built-in graphics, but you can use any graphic that you have available as long as the format is supported.

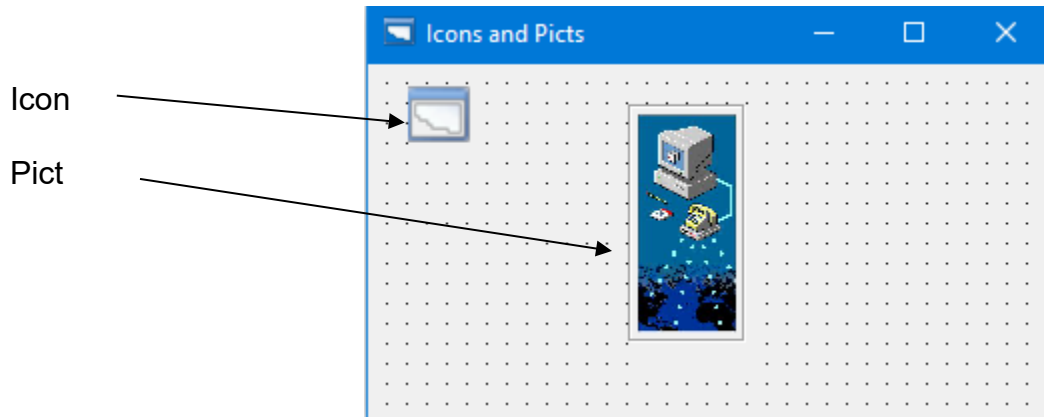
Picts are typically used for dynamic environments like creating graphics on the form at run time. Icons are more efficient and use few resources than Picts.

## STANDARD CONTROLS - continued

### Icons and Picts - continued

While Icons and Picts both render graphic images, each is designed for a specific purpose. An Icon control is best used with an actual icon is to be displayed. For other graphical images, a Pict control is better suited.

Pict controls possess two properties that Icons do not – the **AutoZoom** and **AutoScale** properties. When set to True and Best respectively, the image that is loaded into the object will resize to fit the size of the object's dimensions. This is very useful when sizing very large or small pictures. This sample shows an Icon and a Pict with AutoScale and AutoZoom properties set to True and Best. A JPEG file was used for the Pict resource.



## **EXERCISE – OFFICE EQUIPMENT INFORMATION**

Even though we have not yet discussed coding other than using assignment statements, it is helpful to practice creating user interfaces with the tools we have discussed so far.

This exercise demonstrates a simple use of Radio Buttons, a GroupBox, a Multiline EditText object and an Image control. The only code required is assignment and call statements.

The application has four Radio Buttons in a GroupBox, a Pict control, a StatText object, and a multi-line EditText.

When the user clicks on one of the Radio buttons, the application will load a picture corresponding to the option selected, into the Pict control and display a description of the equipment selected in the EditText object.

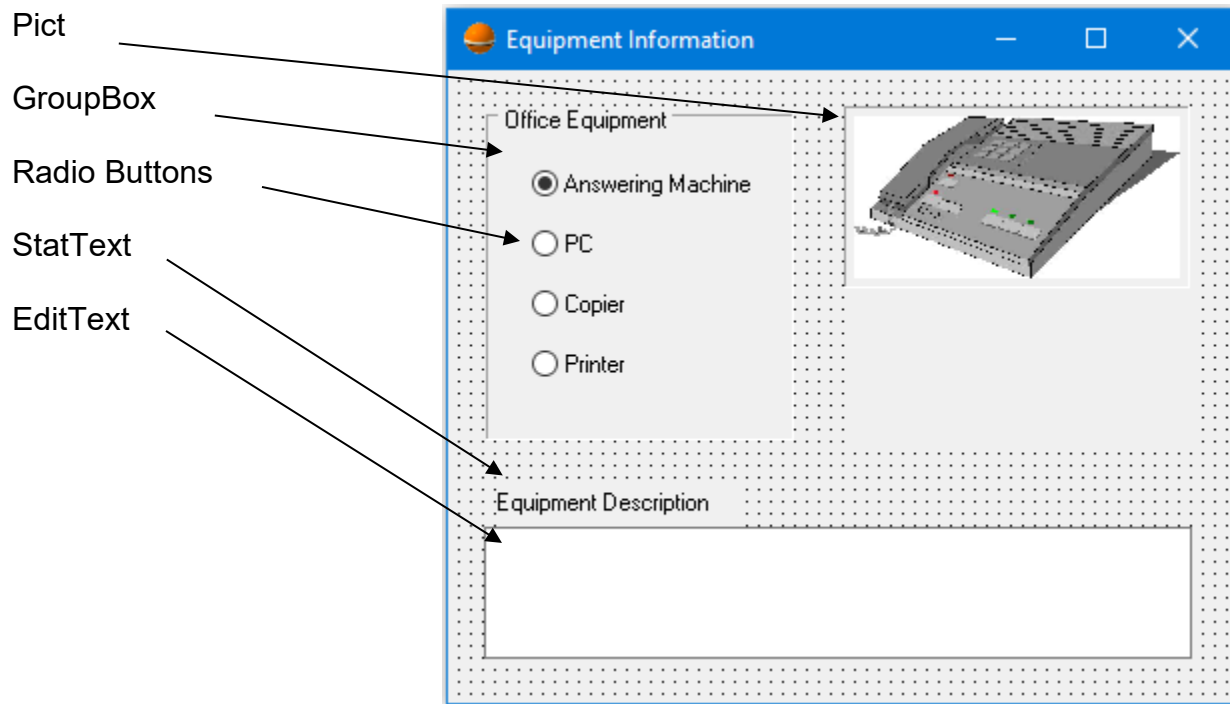
1. Start a new source file (equipment.pls).
2. Add the following lines of code to the source file:

```
MAIN      PLFORM          EQUIPMENT.PLF
.
          FORMLOAD        MAIN
.
          LOOP
          EVENTWAIT
          REPEAT
```

3. Using the Form Designer create a new form (equipment.plf).
4. Set the form's Name to "frmEquipment", the WindowType property to "Primary Fixed", and the Title to "Equipment Information".
5. Draw a GroupBox and change its title to "Office Equipment".
6. Draw four Radio Buttons inside the GroupBox. Change the Names and Titles of the buttons to something meaningful. The prefix "rdo" or "opt" is often used for Radio buttons. Set the GroupID property of all four buttons to one (1) and the Value property of the first one to "1".

**EXERCISE – OFFICE EQUIPMENT INFORMATION - continued**

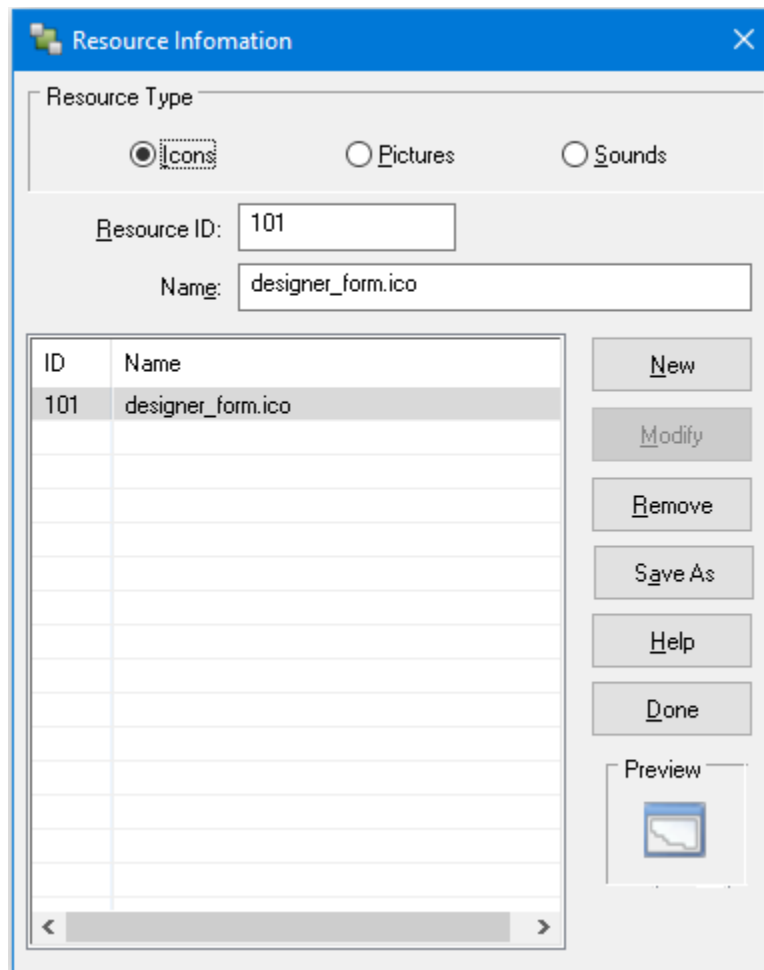
7. Draw a Pict object on the form and set the AutoZoom property to True. Set the Pict's Name to "imgEquipment" and the Border property to True.
8. Draw a StatText and assign "Equipment Description" to the Text property.
9. Draw an EditText under the StatText. Change its name to "txtEquipment". Set its ReadOnly property to True since the user will not type anything into this object. Set MaxLines to something greater than one as appropriate for your design. Also, set Wordwrap to True and clear the Text property.





**EXERCISE – OFFICE EQUIPMENT INFORMATION- continued**

10. Add a STOP instruction to the form's Close event.
11. Next, we need to add four images to the resources of our form. Begin by clicking on Tools in the menu of the Form Designer and then on Resources. The following dialog is displayed:



## Form Designer's Resource Window

12. Add four images by first selecting "Pictures" and then clicking the New button. Your instructor will supply the images.

**EXERCISE – OFFICE EQUIPMENT INFORMATION**

ID	Name
1	Answmach.jpg
2	pcomputer.jpg
3	Copymach.jpg
4	Printer.jpg

Once the four images have been added to the form's resources, click "Done" to close the dialog box.

The click event procedure for each Radio button should load an appropriate picture into the image control. To accomplish this, we need to set the Resource property for the Pict at run time as each Radio button is clicked. Since the Radio Buttons are mutually exclusive, only one picture can be loaded at a time.

```
SETPROP    imgEquipment,Resource=1
```

- Also, each time a Radio Button is clicked we want to add a description in the EditText object. This is done with a statement such as:

```
SETITEM    txtEquipment,0,"This is a "::  
           "digital answering machine"::  
           " valued at $150. It also has"::  
           " a telephone and remote"::  
           "playback capabilities"
```

**EXERCISE – OFFICE EQUIPMENT INFORMATION - continued**

14. Finally, to select the answering machine when the program first starts, add the following line to the form's Load event.

```
CALL      Click_rdoAnswerMachine
```

15. Save your form.
16. Test the program.
17. This completes the exercise.

## **STANDARD CONTROLS - continued**

### **DataList and ComboBox**



DataLists and ComboBoxes support a list of choices from which the user may select. A ComboBox is a combination of an EditText and a DataList.

You use a DataList when you have sufficient room on the screen to display six to eight entries. If you have limited space or if you want to allow the user to enter a value that is not in the list, you would use a ComboBox.

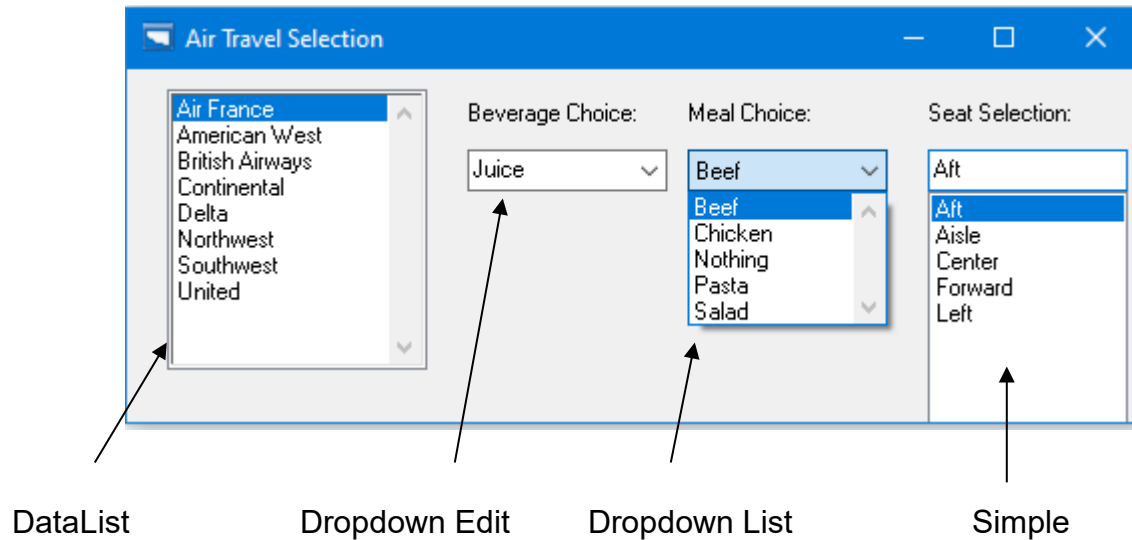
Selection is normally made in a DataList by the user double-clicking the desired selection. You can also allow the user to select more than one item in a DataList by clicking on one and pressing the CTRL or SHIFT keys while clicking another item. This is referred to as "extended selection". If you are allowing extended selection, you normally have the user make their selection and then click a Button to process the selected items.

If there are more items in the list than can be displayed, Windows supplies a scrollbar so that the user can see the remainder of the list. The user can also use the arrow keys to move up or down in the list one item at a time or the PageUp and PageDown to move forward and backward by as many items as visible in the DataList.

ComboBoxes do not support extended selection. Selection is made by dropping down the list and clicking on an item in the list. This places the selected item in the text portion of the ComboBox (at the top). Finally, the user will normally click a Button to process the selection.

## STANDARD CONTROLS - continued

### DataList and ComboBox - continued



There are three styles of ComboBoxes defined by the Style property.

Dropdown Edit is a drop-down ComboBox that allows the user to select from the list or type in his own selection in the text portion at the top of the ComboBox.

A Dropdown List style ComboBox only allows selection from the list.

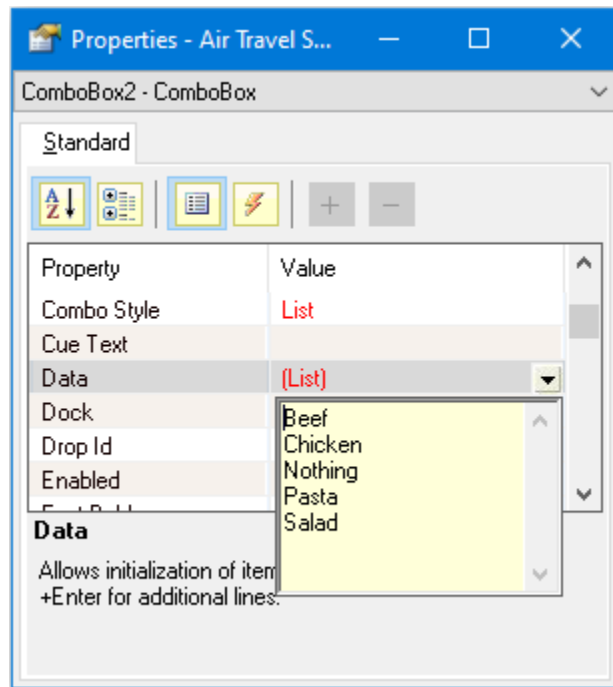
A Simple ComboBox behaves much like a DataList except that it also allows user entry in the text portion at the top of the ComboBox.

DataLists allow selection only and do not allow user entry.

## STANDARD CONTROLS - continued

### DataList and ComboBox - continued

The Data property can be set at design time for either the DataList or the ComboBox. To add items at design time, type each entry followed by the CTRL+ENTER key. The maximum number of items in the list is approximately 32,767.



Items may also be added at run time using the AddString method. For example:

```
List1.AddString Using "New Item"
```

The Sorted property may be set to True allowing Visual PL/B to maintain the list in alphabetical sequence regardless of how they were entered. The sequence is always ascending and cannot be altered.

## **STANDARD CONTROLS - continued**

### **DataList and ComboBox – continued**

The GETITEM instruction returns both the selected item number and the string associated with a particular item. The normal process for working with DataLists and Comboboxes is to retrieve the selected item number and then the string associated with that item.

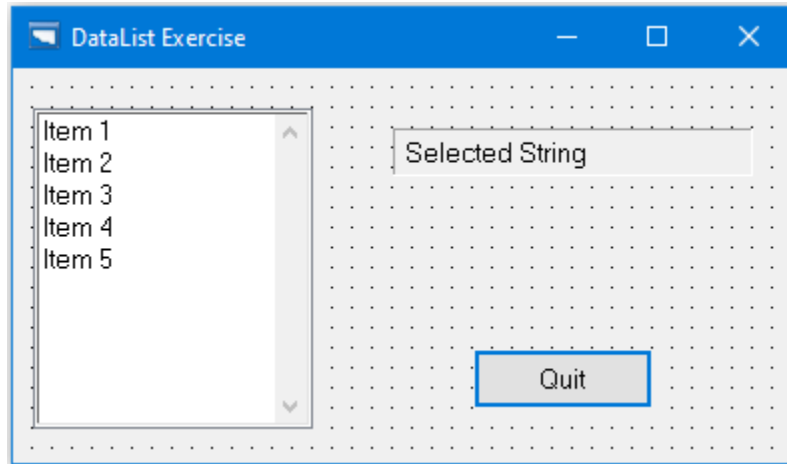
```
INDEX      INTEGER      4
STRING     DIM          30
LIST1      DATALIST
*
. Retrieve the selected item number
.
      GETITEM    LIST1,0,INDEX
*
. Retrieve the selected item string
.
      GETITEM    LIST1,INDEX,STRING
```

There are many other properties and methods that are available at run time for DataLists and ComboBoxes. We will explore these in subsequent chapters.

## **EXERCISE – DATALISTS**

This exercise provides exposure to the DATALIST control.

1. Create a form and add a DATALIST and a STATTEXT object.
2. Fill the DATALIST during the form load event with records from a text file.
3. When the user selects a DATALIST item via double-click, retrieve the selected item and place it into the STATTEXT object.
4. Experiment with the SORTED property of the DATALIST object.



**Sample Form**



## STANDARD CONTROLS - continued

### Shape and Line Tools



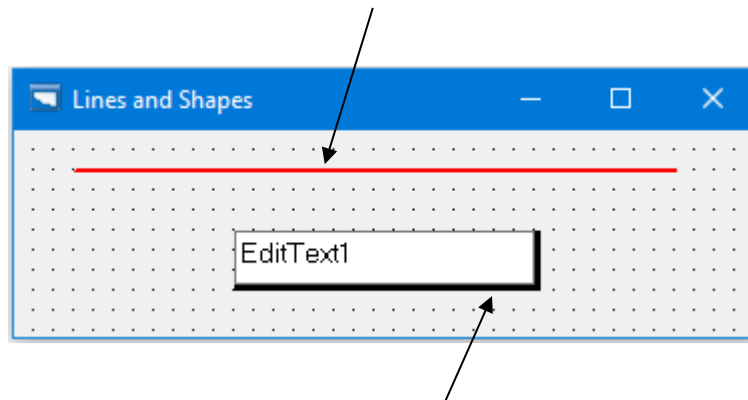
The Shape tool creates graphics on your form. The Shape tool by default creates a rectangle on the screen. By setting the Shape property, you can create circles, ovals, squares, rounded rectangles, and rounded squares.

The Line tool draws lines on the form. The lines may be solid or patterned and have any width or color desired. Patterns are only in effect when the width is one.

These tools add simple graphics to a form and do not respond to any events. You may use them to enhance the appearance of your user interface.

Rather than a top, left, height, and width set of properties, lines use X1,Y1,X2, and Y2 to denote the starting and ending positions.

A line can separate areas of a form.



A Shape can make a shadow behind an EditText.

## **STANDARD CONTROLS - continued**

### **Shape and Line Tools- continued**

The properties for the Shape and Line tools are listed below

**Border Color** and **Fill Color** – Set to desired edge and interior colors using the color palette.

**Border Pattern**– Controls how the line or outline of the shape is drawn

- Dash
- Dash-Dot
- Dash-Dot-Dot
- Dot
- Inside Frame
- Solid

**Border Width** – Allows specification of the width of the line or outline of the shape. If you set it to something other than one, you cannot use any of the dashed or dotted Border Pattern values.

**Fill Style** – Determines the interior pattern of the shape.

- Cross
- Diagonal Cross
- Downward Diagonal
- Horizontal Line
- Solid
- Transparent
- Upward Diagonal
- Vertical Line

**Shape** – Determines the type of a shape control.

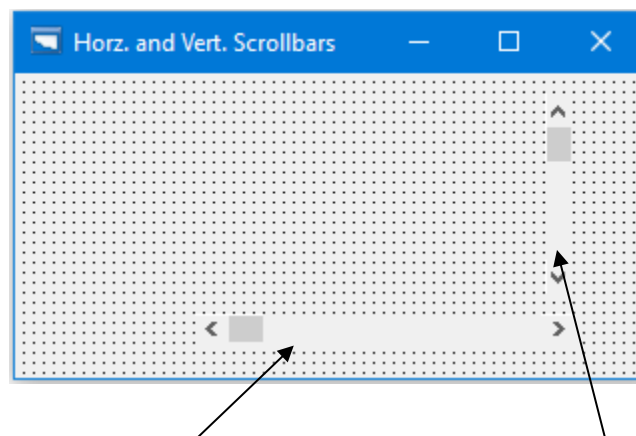
- Oval
- Rectangle
- Round Rectangle

## STANDARD CONTROLS - continued

### HScrollBar and VScrollBar



The horizontal and vertical scrollbars allow scrolling for objects that do not normally provide a scrollbar. They can also be used in conjunction with other controls to allow user input.



**Horizontal Scrollbar**

**Vertical Scrollbar**

Most objects that need scrollbars supply their own. The Horizontal and Vertical scrollbars are provided for unique situations.

## **STANDARD CONTROLS - continued**

### **Timer**



The Timer control is useful if you want to have a certain event occur on a regular, timed basis. For example, you may want to execute a process at a specific time interval.

Timers are not very accurate in small dimensions, but they work well for longer time frames. The system generates 18 clock ticks per second. Even though the interval is measured in milliseconds, the precision is no more than 1/18<sup>th</sup> of a second. Checking the system clock is more common when accurate timing is required.

When you place a Timer on a form, the value defined by the Timeout pseudo property determines how often the Timer event is triggered. The Interval can be between zero and 64,767 (about 65 seconds) and represents tenths of a second. Place the code that you want to be executed in the Timer event and when the time interval is reached, the processing will initiate.

The StartTimer pseudo property instructs the object to begin timing when the form is loaded.

Timer controls are invisible at run time.

**STANDARD CONTROLS - continued****Timer - continued**

To use Timers for periods longer than supported by the control, simply use a counter as part of the event routine.

```
Timer1      Timer
Count       Form      "5"
Result      Integer    4
.
      CREATE      Timer1,600                      // one minute timer
      ACTIVATE    Timer1,TimeCheck,Result
.
      (program code)
.
      STOP
*
.The timer has expired
.
TimeCheck
      DECR        COUNT
      RETURN      IF NOT ZERO
.
      (do something)
.
      MOVE        "5",Count                      // reset the count
      RETURN
```

**EXERCISE – Timers**

This exercise provides exposure to the Timer control.

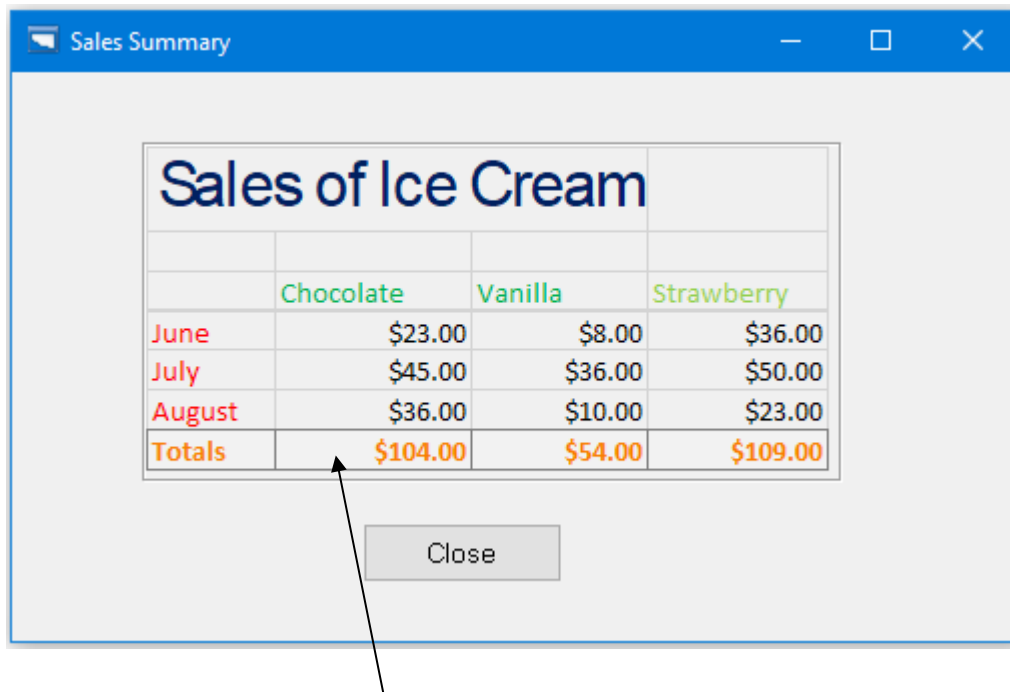
1. Create a form.
2. Using a timer, the program should display a text string in a StatText after five seconds.
3. After two seconds of viewing the string, erase it.
4. Continue displaying and erasing the string until a Quit button is clicked.

## STANDARD CONTROLS - continued

### OLE Container



The OLE Container embeds objects from other applications. You can use this control at run time to allow the user to edit the data using the application that created it. For example, you may have a sales application that displays an Excel spreadsheet showing sales by region. When the user double-clicks the OLE control at run time, he will see the spreadsheet loaded in Excel ready for him to edit.



OLE Container control with an Excel spreadsheet loaded.

## **STANDARD CONTROLS - continued**

### **OLE Container Properties**

**Auto Activate** - This property defines when an OLE container should be activated. Options include:

- When the container object is double-clicked or the ENTER key is pressed while the container object has the focus. (default)
- when the container object gets keyboard focus.
- Not automatically activated. It can be activated by using the \$DoVerb method.

**Auto Verb** – This property controls the automatic context menu for an OLE insertable object. The menu contains any verbs that the OLE insertable object provides.

**Display Type** – The Display Type specifies how the OLE insertable object is drawn within an OLE CONTAINER object. Valid options include:

- The content of a OLE insertable object is shown. (default)
- The OLE insertable object is shown as an icon.

**Size Mode** – This property controls how the OLE insertable object is displayed within an OLE CONTAINER object. Valid options include:

- resized to the size of the OLE insertable object.
- clipped to the size of the OLE container object window. (default)
- evenly scaled for the best fit in the OLE container object window.
- evenly scaled for the best fit in the OLE container object window.



## **STANDARD CONTROLS - continued**

### **OLE Container Properties - continued**

**Source Doc** – This property specifies a file name opened or created by the OLE insertable object.

**Update Options** – This property controls when the contents of an OLE insertable object are updated within an OLE CONTAINER object. Valid options are:

- Any time the data is changed. (default)
- When the linked data is saved from within the application in which it was created.
- When the update method is called.

### **OLE Container Methods**

**\$Delete** – The \$Delete method removes the OLE insertable object from the container.

**\$Update** - Modifies an OLE insertable object.

**\$DoVerb** - Performs an action for an OLE object. Valid actions are:

- OLEPrimary - The default action for the object.
- OLEShow - Activates the object for editing.
- OLEOpen - Opens the object in a separate application window.
- OLEHide - For embedded objects, hides the application that created the object.
- OLEUIActivate - If the object supports in-place activation, activates it and shows any user interface.
- OLEInPlaceActivate - If the user moves the focus to the OLE container control, creates a window for the object and prepares the object for editing.
- OLEDiscardUndoState - When the object is activated for editing, discards all record of changes that the object's application can undo.

**EXERCISE – OLE Containers**

This exercise provides exposure to the OLE Container control.

1. Create a form and add an OLE Container object.
2. Link the OLE Container object to the Wordpad application.
3. Test the application.
4. Experiment with the AutoActivate and other properties.

## **STANDARD CONTROLS - continued**

### **Animate**



The Animate control allows the addition of animated graphics to your form. These controls are generally shown while waiting for some lengthy process to complete. Most Windows users are familiar with animation used by operations such as copying and moving files within the File Explorer. Use these controls only after careful thought and consideration.

The source file for an Animate object is an AVI file. This file is a series of frames like a movie. An AVI file contains no sound. The size of the images is controlled within the file.

#### **Animate Properties and Methods**

**Auto Play Property** – This property controls the automatic playing of an AVI file when form is loaded.

**Open Method** – The Open method associates a disk file with the Animate object. If AutoPlay is enabled, the animation will begin.

**Play Method** – The Play method begins the animation display.

**Stop Method** – The Stop method halts the animation display.

## **STANDARD CONTROLS - continued**

### **MRegion**



A MRegion control defines a portion of the form where mouse events can be captured. As most controls process their own mouse events, this control is provided to allow the capture of events in areas without controls.

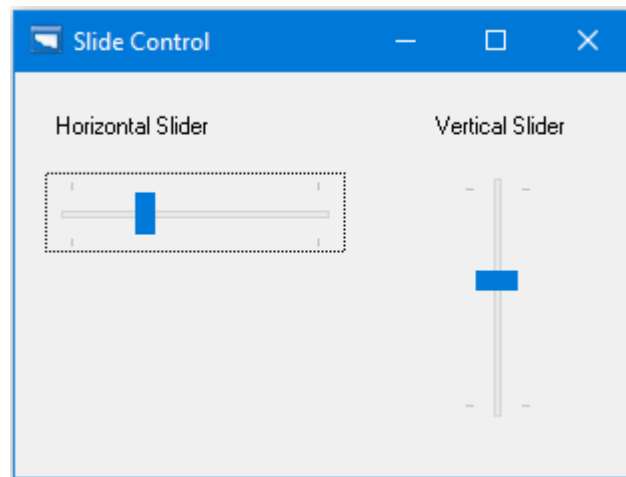
A MRegion has no visible component and only a basic set of properties to define its position and its ability to receive events.

## **STANDARD CONTROLS - continued**

### **Slider**



A slider is control equipped with a small bar, also called a thumb that slides along a visible line. There are two types of sliders: horizontal and vertical:



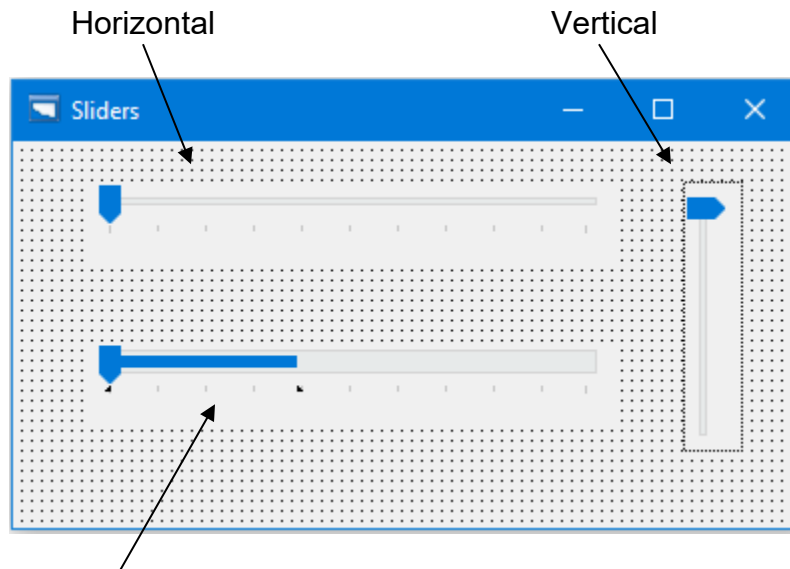
To use the slider control, the user can drag the thumb in one of two directions. If the slider is horizontal, the user can drag the thumb left or right. The thumb of a vertical slider can be dragged up or down. This changes the position of the thumb. The user can also click the desired position along the line to place the thumb at the desired location.

Alternatively, when the slider has focus, the user can also use the arrow keys of the keyboard to move the thumb.

## **STANDARD CONTROLS - continued**

### **Slider - continued**

A slider is configured with a set of values from a minimum to a maximum. Therefore, the user can make a selection included in that range. Optionally, a slider can be equipped with small indicators called ticks:



Horizontal with Selection Area

The ticks can visually guide the user for the available positions of the thumb mark. A slider can be used to let the user specify a value that conforms to a range. When equipped with ticks, a slider can be used to control exact values that the user can select in a range, preventing the user from setting just any desired value.

### **Properties**

**Orientation** – Defines the slider style as horizontal or vertical.

**Min, Max** – Specifies the range of the slider.

**SelStart, SelRange, SelLength** – Defines and enables a selection area.

**TickStyle** – Sets the position of the tick marks – top, bottom, right, left, both, or none.

**Tick Frequency** – Controls the number of tick marks displayed.

## STANDARD CONTROLS - continued

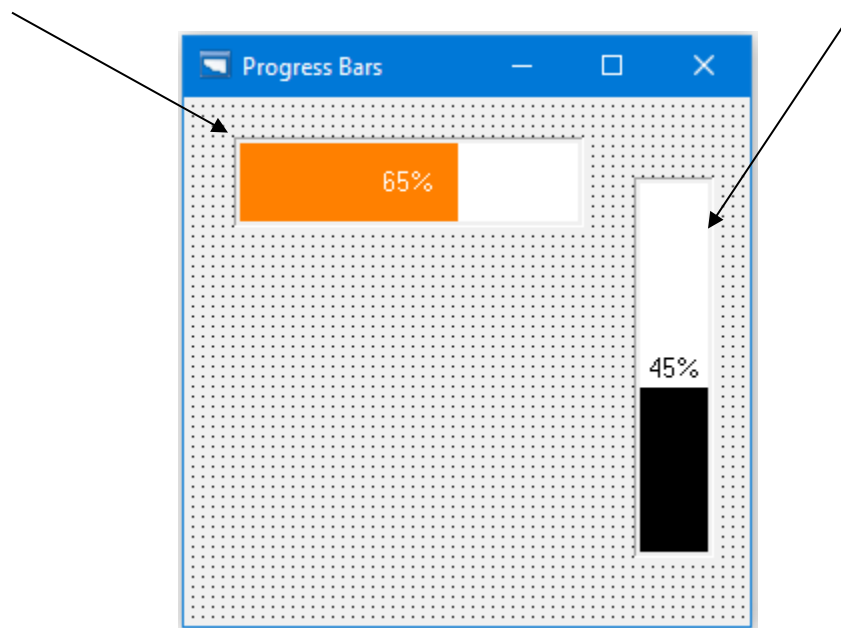
### ProgressBar



The Progress control displays the evolution of an activity, especially for a long operation. Like a label control, a progress control is used only to display information to the user who cannot directly change it. A progress bar may be horizontal or vertical. Its orientation is defined by its shape.

Horizontal Progress Bar

Vertical Progress Bar



A progress bar displays regular small rectangles inside a longer or taller rectangle. This outside rectangle serves as their border. There are several options for the border style.

#### Progress Properties

**Background Color** – Defines the color of the unused portion.

**Foreground Color** – Defines the color of the used portion.

**Show Percentage** – Controls whether the percentage is shown.

**Style** – Controls the border type.

**Value** – Defines the progress amount.

## **STANDARD CONTROLS - continued**

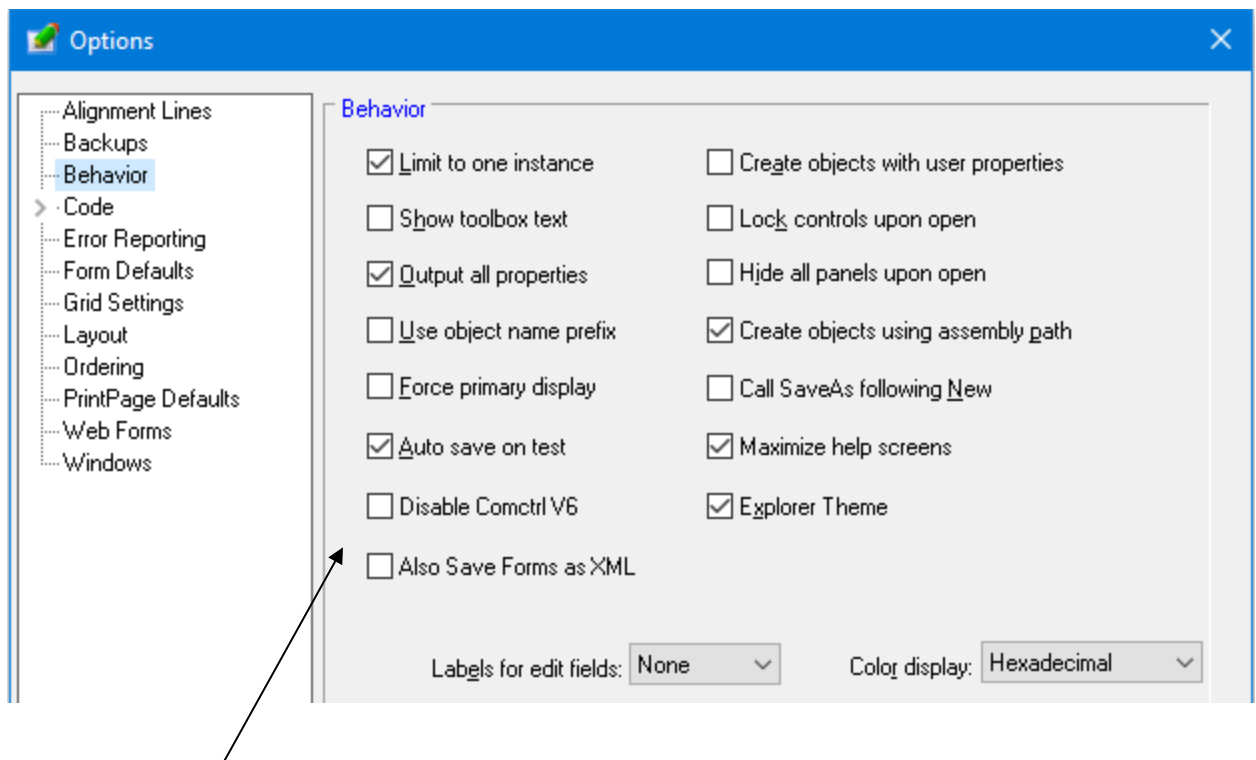
### **Panel**



A panel is a container much like a form or window. The primary difference is that a panel is a sub-class of a window and must have a window or another panel as its parent. The panel itself serves as the parent for objects placed on it.

Panels are handy anytime you need to divide the window into segments. Since the panel is the parent for objects placed on it, actions to the panel such as toggling visibility, effect all objects on the panel. In that respect, a panel serves as a collection just as a form does.

The Designer makes use of Panels in its Options window. As categories are clicked in the TreeView object on the left, the active panel is hidden and the selected property is shown. This is accomplished by simply setting the Visible property of the Panels.



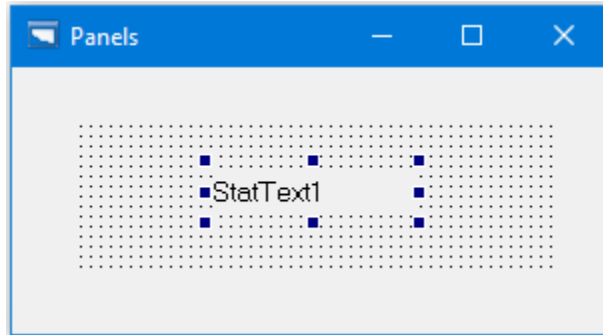
**A Panel with Objects**



## **STANDARD CONTROLS - continued**

### **Panel - continued**

Once a has been created, objects may be placed on the panel using the Designer. A grid appears on the panel during the drawing process and when a panel's child object is selected.



**An Object on a Panel**

To implement a form such as the Options window shown previously required multiple panels. The panels should have the same top, left, height, and width properties.

After creating the first panel in the designer, right click on the panel in the design window or in the Outline to display the shortcut menu. The first item in the menu is "Hide". Hide simply make the panel invisible at design time. The form is then clear and ready for the addition of the second panel. To make a hidden panel visible, right click the panel in the Outline and select "Show".

#### **NOTE**

To avoid confusion, you should only have one Panel visible at a time. Before showing another panel, always hide the current panel.

## **STANDARD CONTROLS - continued**

### **Splitter**



A Splitter allows the user to resize docked controls. The Splitter control is often used on forms with controls that have varying lengths of data to present, like File Explorer, whose panels contain information of varying widths at different times.

To illustrate a Splitter, do the following:

1. Create a new form
2. Add a Datalist
3. Set the Datalist's Dock property to Left
4. Add a Splitter
5. Test the form.

When the mouse cursor is placed over the right edge of the Datalist, the split cursor is shown. Pressing the left mouse button down and dragging to the right will widen the Datalist.

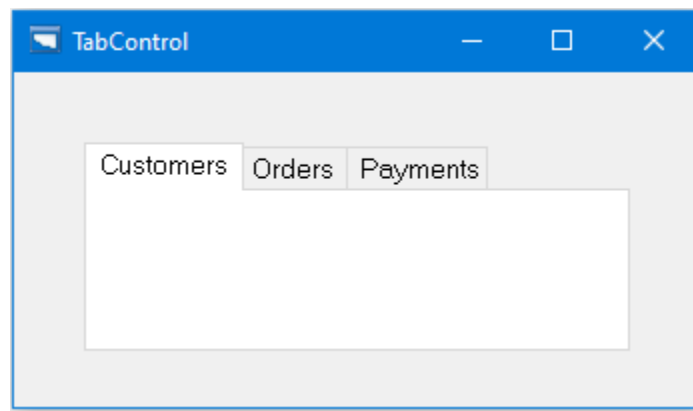
An example of the use of a Splitter can be found in the IDE. The source map and the message window both employ a Splitter control.

## **STANDARD CONTROLS - continued**

### **TabControl**



The TabControl displays multiple tabs, like dividers in a notebook or labels in a set of folders in a filing cabinet. The control itself is a simple frame with a row of tabs. The tabs each have labels and the tabs may be arranged on any side of the frame.



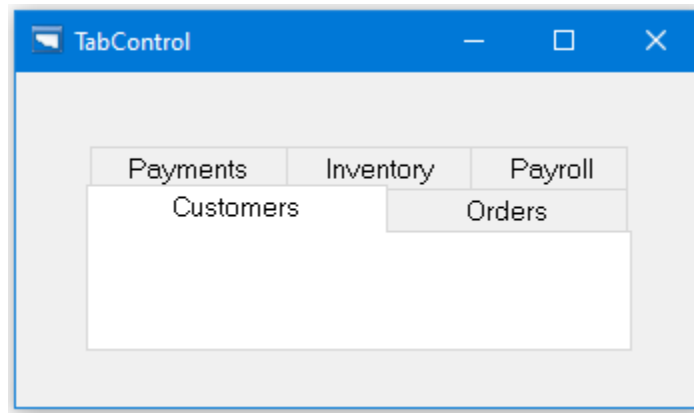
**A Simple TabControl**

The purpose of the TabControl is simply to display the labels inform the program when a label is clicked. The program code must take whatever action is to occur such as hiding one set of controls and showing others. For that reason, TabControls normally employ Panels to aid in the management of the controls they display.

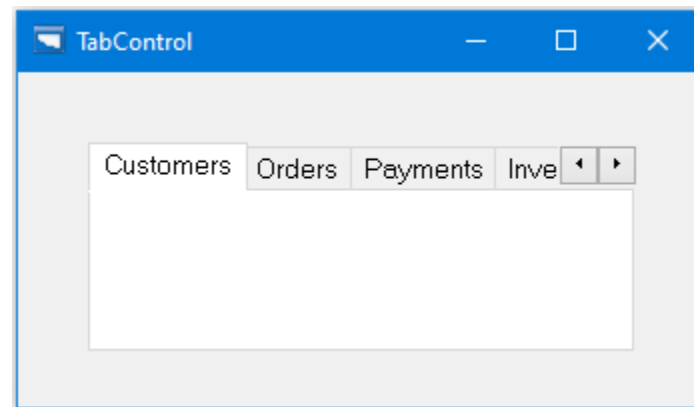
## **STANDARD CONTROLS - continued**

### **TabControl - continued**

The control supports multiple rows of tabs by setting the MultiRow property to True. If MultiRow is False and the width of the labels exceeds the width of the TabControl, a set of arrows at the end of the label list allow scrolling the labels.



**MultiRow Labels**



**Scrolling Labels**

## **STANDARD CONTROLS - continued**

### **TabControl – continued**

To create a TabControl and its associated panels, use the following technique:

1. Create a TabControl.
2. Set the TabLabels properties for the labels desired.
3. Create the first Panel in the client area of the TabControl.
4. Add the required controls to the Panel.
5. Hide the first Panel.
6. Create the next Panel and required objects.
7. Set the new Panels Visible property to False.
8. Repeat steps 5 - 7 until all Panels have been created.

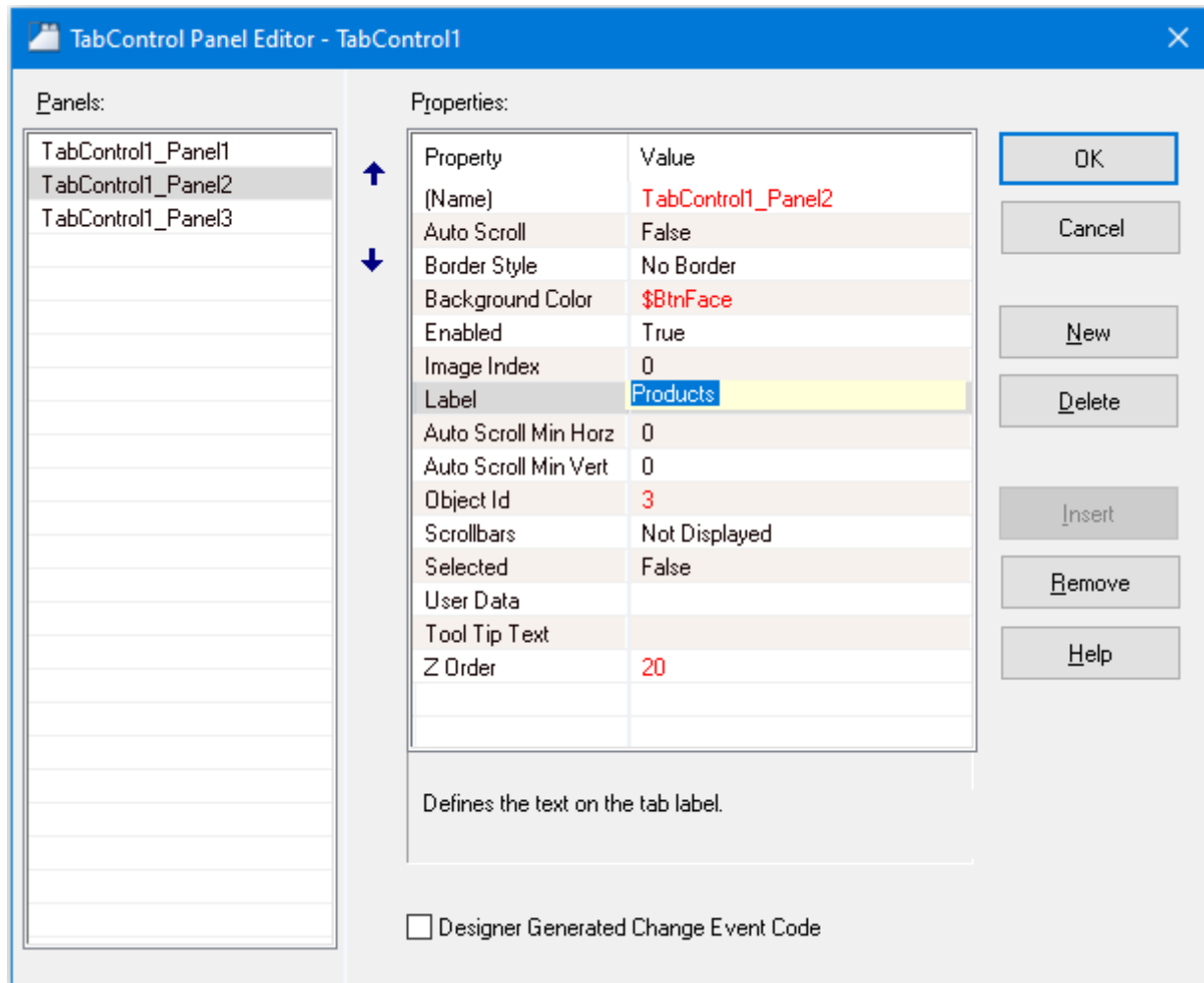
The code to hide and show the appropriate panel at execution time can be written in the Change Event as follows:

```
*
.Disable all panels
.
    SETPROP    Panel1,Visible=0
    SETPROP    Panel2,Visible=0
    (continue for other panels)
*
.Show the selected panel
.
    SWITCH     #EventResult
    CASE       1
    SETPROP    Panel1,Visible=1
    CASE       2
    SETPROP    Panel2,Visible=1
    (continue for other panels)
    ENDSWITCH
```

## STANDARD CONTROLS - continued

### TabControl – continued

A feature of the Designer aids in the creation and management of panels for TabControls. By selecting the “TabPanels” property, a special panel editor is displayed.



**The TabPanel Editor**

## **STANDARD CONTROLS - continued**

### **TabControl – continued**

When using the TabPanel Editor any previous defined TabLabel property setting and TabControl Change Event code will be discarded and replace with values from the editing session.

When removing panels, a warning will be displayed if the Panel has object that will be lost when the panel is deleted.

The order of panels may be changed by the up and down arrows between the panel list and the properties.

The Insert function allows a previously created independent panel to be associated with the tab control. Remove make an associated TabControl panel and independent panel.

**EXERCISE – TabControl**

This exercise provides exposure to the TabControl object. Please perform the following WITHOUT using the TabPanels property.

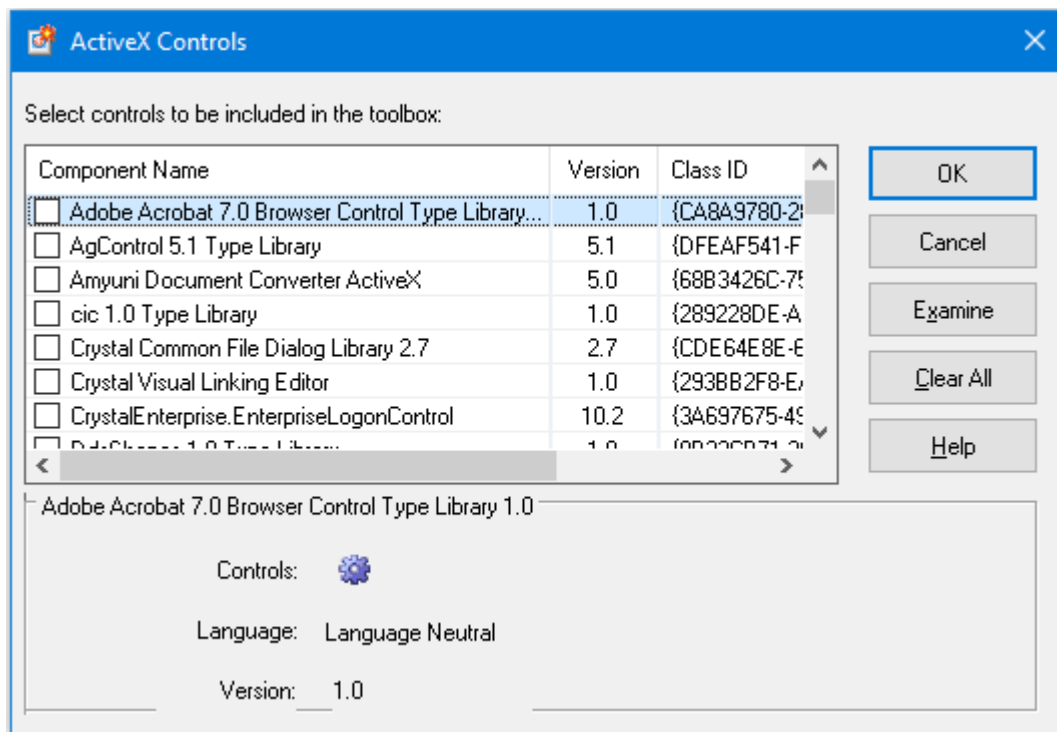
1. Create a TabControl.
2. Add two or more labels to the control.
3. Create a panel for each label defined.
4. Add objects to each panel as desired.
5. Insert code to change panels as tabs are selected.



## **ACTIVEX CONTROLS**

Many ActiveX controls are available today. Some are available with the basic Windows operating system and others are available via the Internet. Using ActiveX controls in your program is almost as easy as using the standard controls. Once an ActiveX control is placed in the Toolbox, you use it as you would a standard control. You may draw it on the form, set properties, and write event code.

To include any of the ActiveX controls in your form, select Tools and Add Control from the Form Designer's menu. Alternately, you may right click on the Toolbox and select "Add Control" from the shortcut menu. The Add Control dialog will then be displayed.



**Add Control Dialog**

This dialog lists all the controls registered on your PC. Windows provides some of the controls while others were added by applications installed on the PC. Many third party vendors supply ActiveX controls that may be used with Visual PL/B.

## **ACTIVEX CONTROLS - continued**

Before using an ActiveX control, it must be registered. This is normally done by the control's installation process.

Be aware that any ActiveX control used in your application must be present on any computer that wishes to run your program. Some controls have licensing and use restrictions. Read the documentation for any ActiveX controls you plan to use carefully.

You can move ActiveX controls to and from the Toolbox using the following procedures:

### **To add an ActiveX control to the ToolBox:**

1. From the Tools menu, select Add Control. Alternatively, you may right click on the Toolbox and select "Add Control".
2. Click on the control to be added from the list of registered controls
3. Click on the OK button

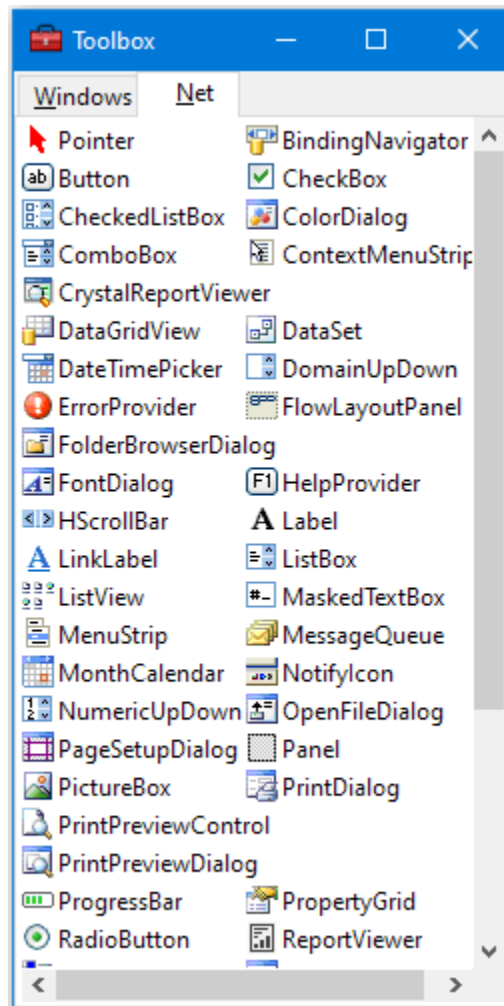
### **To remove an ActiveX control from the Toolbox:**

1. Remove all references to the control on all forms. While references remain, the delete control function will not be available.
2. Click the control in the Toolbox.
3. Right-Click the control in the Toolbox.
4. Chose "Delete Control" from the shortcut menu.

## .NET CONTROLS

When running under the PLBNET runtime, the designer allows creation of .Net controls. The use of a NetControl in the Designer is like standard object. Net controls are drawn, sized, moved, deleted, properties are set, and code is written to selected events.

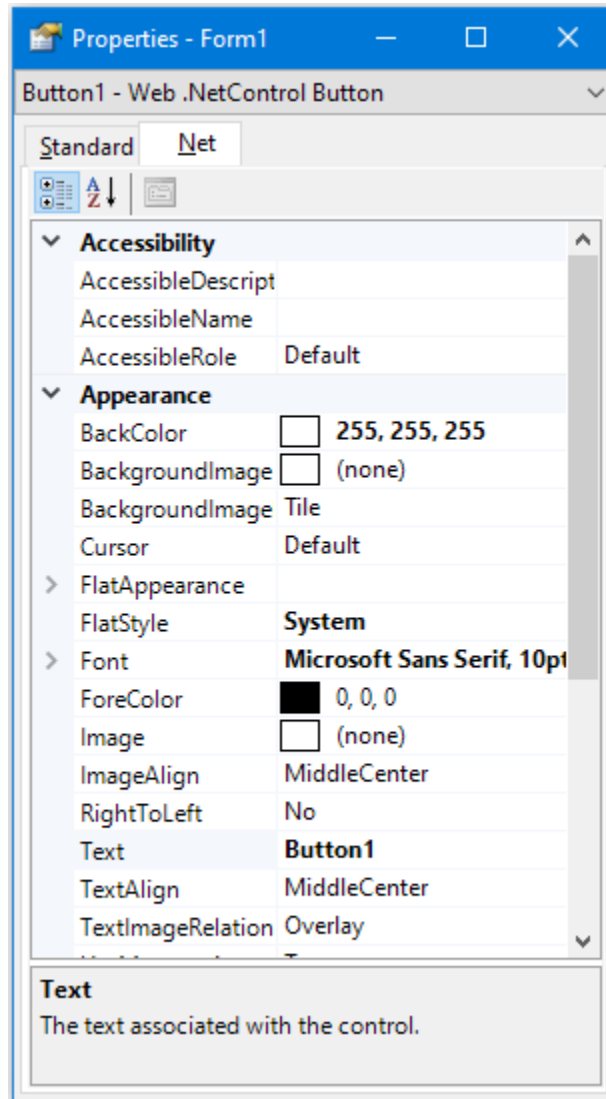
Two areas will appear slightly different with this runtime. First, the Toolbox will have a second tab labeled Net. It will show all the installed .Net objects.



**.Net Toolbox**

## .NET CONTROLS - continued

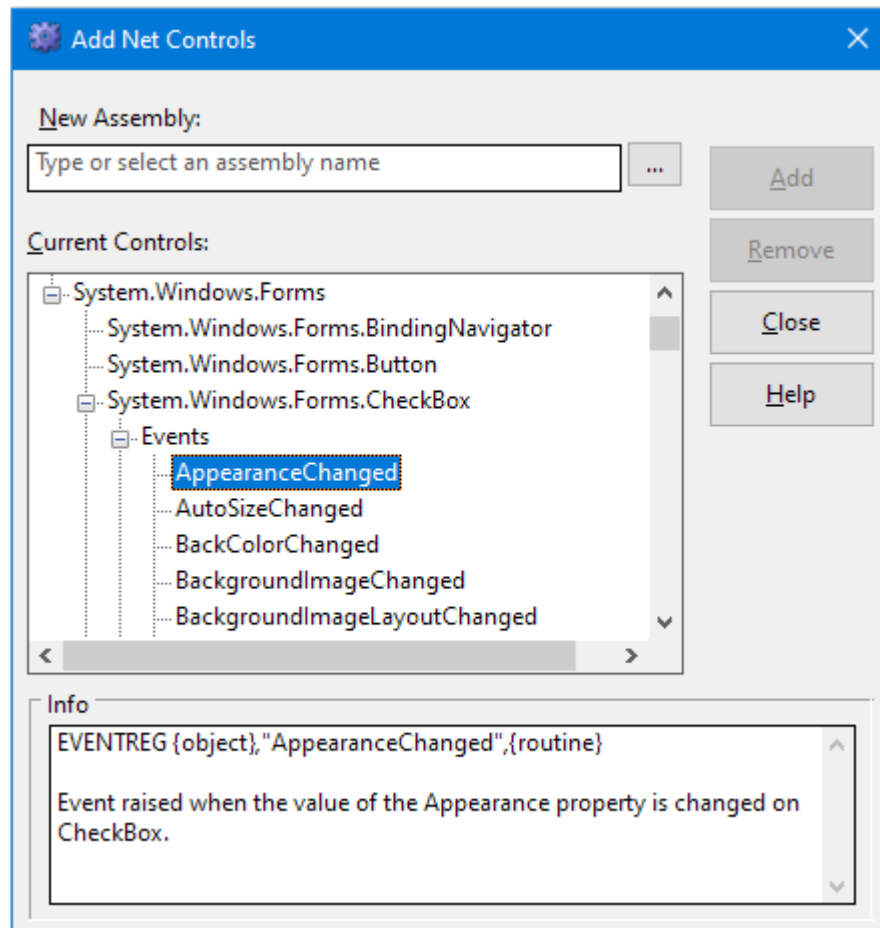
The second change will be to the property window. When a .Net control is selected, a new Net tab will appear at the top of the property window. Selecting the Net tab will produce a .Net Property Grid. This grid is populated with properties from the object itself using a technique called reflection.



**NetControl Property Grid**

## .NET CONTROLS - continued

By default, only the controls that are a part of the System.Windows.Forms assembly are shown. Additional controls are added to the toolbox by selecting Tool and Add Net Controls.



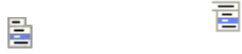
**Add Net Controls Dialog**

The Add Net Controls dialog serves two purposes.

1. New controls may be added by specifying the assembly name or browsing for the assembly and clicking Add. Once added, the control will be visible in the toolbox on the Net tab.
2. Selecting a control allows the properties, events, and methods of the control to be displayed.

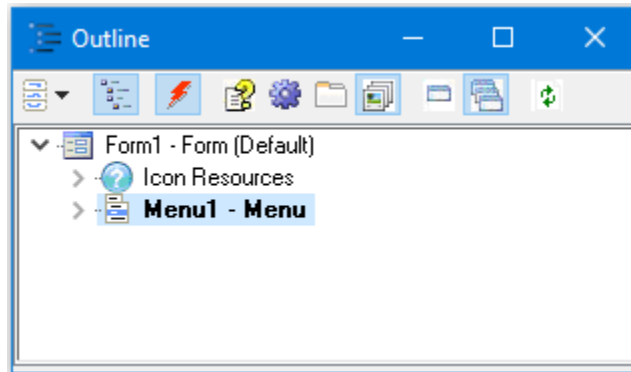
## **ADDING MENUS TO FORMS**

### **Menu, and SubMenu**



To add a menu to a form, begin by drawing a menu object anywhere on the form. Once this is done, a menu line will appear in the Form Outline window with one menu item "Menu1". Further modification of the menu requires the use of the Outline window to select the menu.

Open the Outline window by selecting "Window" and then "Outline" from the Designer's menu. Alternate methods of opening the window include pressing the F6 key or right clicking the form and selecting "View Outline" from the shortcut menu. Once opened, clicking the plus signs (+) will open the various levels of the tree.



**Outline Window**

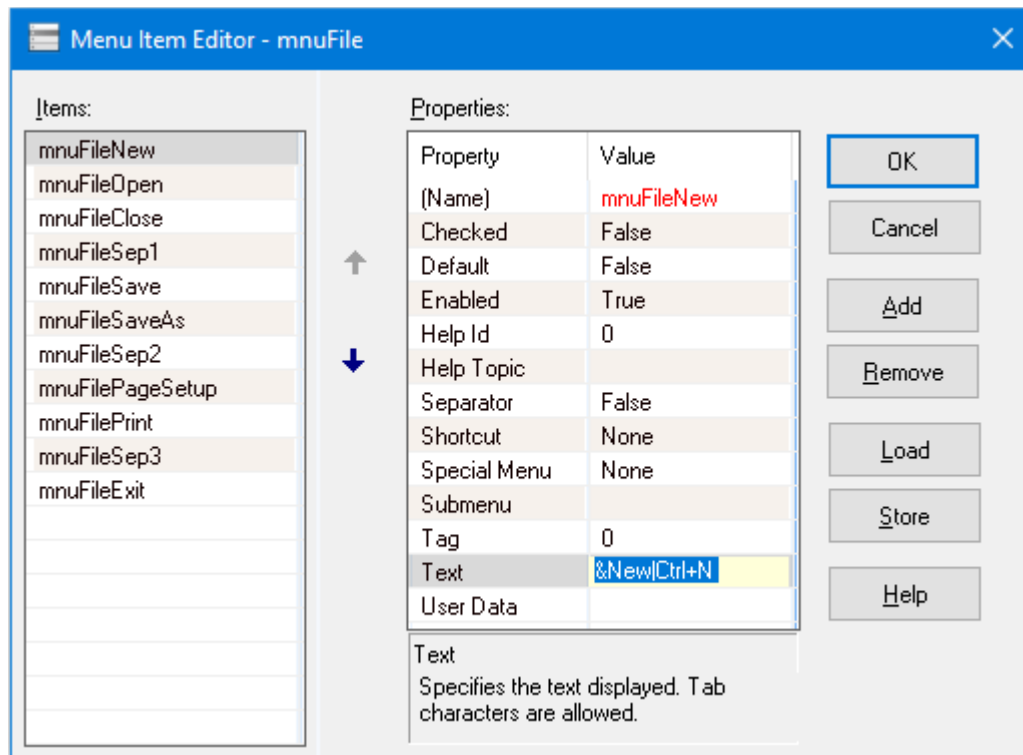
The Outline window displays all objects associated with a form in a hierarchical tree structure. The menu object just added is present with a single submenu entry ("Menu1"). Using the Outline window and the Properties window, a custom menu may be defined for the application.

To change the top-level text displayed with a menu item, click the item in the Outline window and change the Text property in the Properties window. You will normally change the Name property as well before writing event code. Menus typically use a prefix of "mnu" followed by a descriptive string for their name. For example, "mnuFile" should be the File menu.

Each selection within a form's menu bar is a menu item. If your form has a main menu of "File", "Edit", "Format", "Tools", and "Help", you will require five menu items.

To enable the Alt Key menu selection capability, precede the desired letter in the menu text property with the ampersand (&). Note that a control panel display setting normally hides the underscore until the Alt key is pressed (Display/Appearance/Effects).

To add items to a menu, select "Items" within the Property window and click on the ellipses button (...). This action will open the Menu Items Collection Editor where items are maintained.



**Menu Items Collection Editor**



## **ADDING MENUS TO FORMS - continued**

This dialog allows quick entry and maintenance of menu items. To add an item, click "Add" and then set the item's name and text in the Properties pane. It is suggested that a naming convention be followed just as with other controls. In the example above, the menu is named "mnuFile" and each of the items within the menu use that name as a prefix.

Items may be arranged within the menu by selecting the menu item and then using the up and down arrows between the two panes.

Selecting an item and clicking the Remove button will delete a menu item.

The Store button allows the current menu definition to be saved for use in other forms. After clicking the button, simply provide a descriptive name. When editing another form, use the Load function to retrieve the saved form definition. The Designer comes with several common menu definitions already stored.

When all changes are complete, click OK to save the menu or Cancel to abandon the changes.

## **ADDING MENUS TO FORMS - continued**

### **Menu Item Properties**

Each menu item has the following properties:

**Name** – A string used to reference the object in the program code, Properties Windows, and the Outline. Menus and menu items normally have a prefix of "mnu" such as "mnuFile" or "mnuFileSave".

**Checked** – When this property is True, a check mark will appear to the left of the item in the menu.

**Default** – When this property is True, pressing the Enter key will have the same effect as clicking the item

**Enabled** – When set to False, the item will appear gray and will not be selectable by the user

**Separator** – This property defines the item as a separator line. A separator line allows a visual grouping of items in a menu. Any value in the Text property will be disregarded.

**ShortCut Key** – Allows the user to select a menu item using a special key sequence. This action is the same as the user selecting the item using the mouse.

**SpecialMenu** – When set to True, this property associates an Edit menu function such as cut, copy, paste, delete, undo, or select all to the item.

**SubMenu** – This property associates a submenu with the item creating a menu hierarchy.

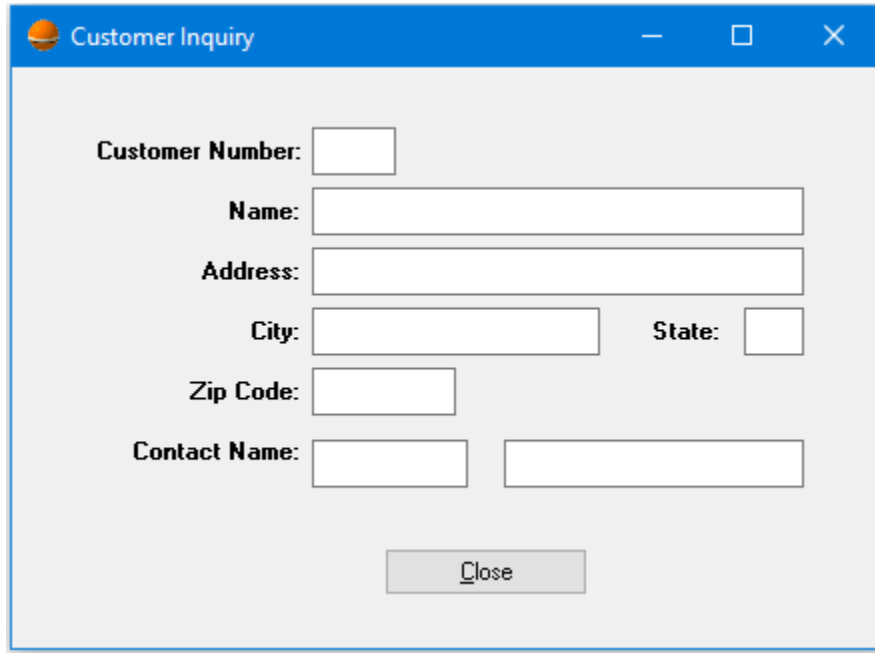
**Tag** – Creates a user-defined string that can be used to identify the item after a button click event.

**Text** – The string displayed for the menu item. To embed a Tab character, use Ctrl+Tab. If you include an ampersand (&) in Text property, it designates the following character in the string as an Access Key. This allows selection of the item by simply pressing the Access Key once the menu is open.

## **EXERCISE – ADD A MENU TO THE INQUIRY FORM**

This exercise provides exposure to the menu object by creating a customer inquiry form.

1. Begin by designing a form as follows:

A screenshot of a Windows-style application window titled "Customer Inquiry". The window has a blue title bar with standard minimize, maximize, and close buttons. The main area is light gray and contains several text input fields. The labels and their corresponding input fields are: "Customer Number:" followed by a small rectangular box; "Name:" followed by a long horizontal box; "Address:" followed by a long horizontal box; "City:" followed by a medium-width box, and "State:" followed by a small box; "Zip Code:" followed by a medium-width box; and "Contact Name:" followed by two adjacent medium-width boxes. At the bottom center of the form is a button labeled "Close".

**Example Form**

## **EXERCISE – ADD A MENU TO THE INQUIRY FORM - continued**

2. Add a File menu with an Exit menu item beneath it.
3. Add an Edit menu with the following menu items and attach the appropriate key sequences:
  - Undo
  - Separator
  - Cut
  - Copy
  - Paste
  - Separator
  - Select All
  - Delete
4. Add a Help menu with a single About item.
5. Specify Access and Quick select keys, as they are normally defined.
6. Code the Exit menu item to terminate the program.
7. Compile and test your application.

The screenshot shows a Windows application window titled "Customer Inquiry". The window has a standard Windows title bar with a blue background and a yellow icon. Below the title bar is a menu bar with three items: "File", "Edit", and "Help". The main area of the window is light gray and contains a form with the following fields:

- Customer Number:** A single-line text input field.
- Name:** A single-line text input field.
- Address:** A single-line text input field.
- City:** A single-line text input field.
- State:** A single-line text input field.
- Zip Code:** A single-line text input field.
- Contact Name:** Two single-line text input fields side-by-side.

At the bottom center of the form is a button labeled "Close".

## **MORE STANDARD CONTROLS**

### **ImageList and ToolBar**



An ImageList control is a repository of images that make working with ListView, TreeView, and ToolBar objects much easier. The ImageList has no visible component and supports no user interaction.

Once an ImageList has been added to your form, you should first define the size. All images with the ImageList must be the same size. The ImageSizeH and ImageSizeV properties define the size.

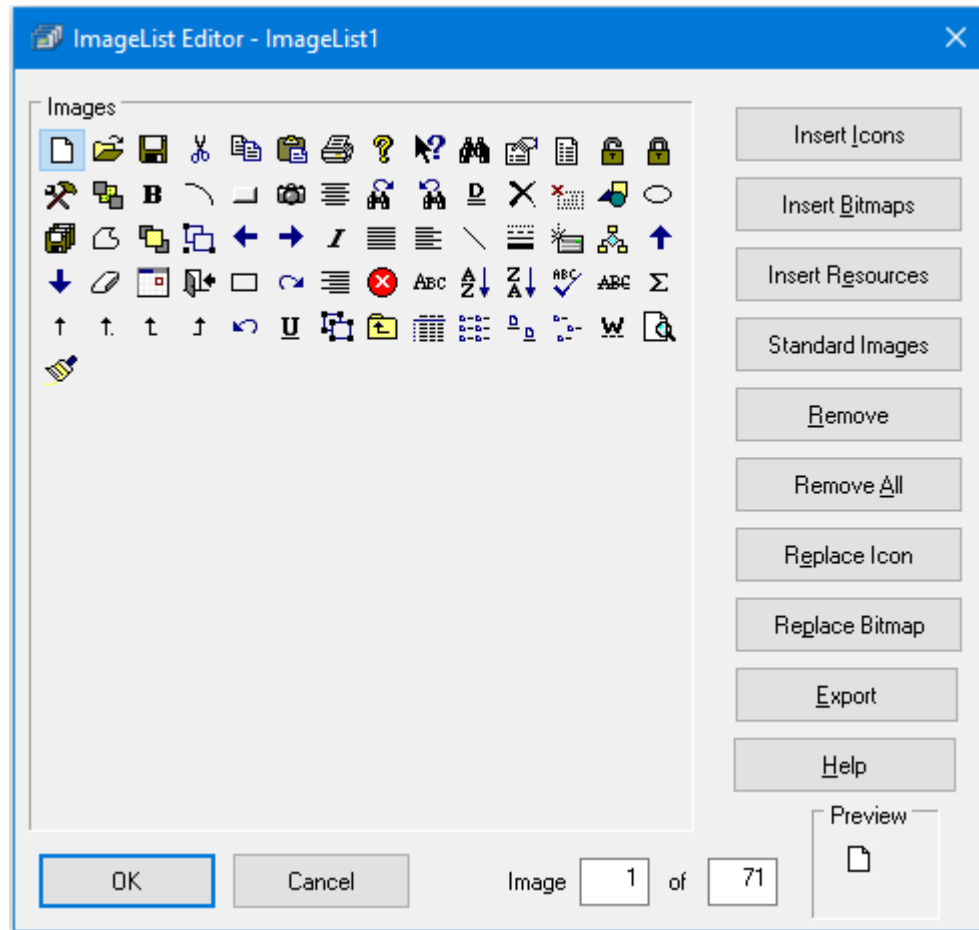
Once the size has been defined, you must add the images. The images may be either icons or bitmaps. Commonly used bitmaps are included in the PLB runtime and may be loaded as a group using the LoadStdToolBitMap method. Methods also exist to remove and replace images and to return the count of images in the ImageList.

To add images to the ImageList, one method is to use the AddBmp or AddIcon method. The method calls are normally place in the forms “FormInit” event. This will initialize the ImageList when the form is initialized. The bitmaps or icons added may be individual files or resources within the form.

## MORE STANDARD CONTROLS - continued

### ImageList and ToolBar - continued

The Designer provides a simple way to manage the images in an ImageList. By selecting the “Images” property and click on the ellipses, an ImageList editor is opened.



**The ImageList Editor**

The ImageList Editor greatly simplifies management of the ImageList.

### ImageList Properties

**ImageColors**— This property specifies the number of bits supported for colors in an ImageList. The default value is 32.

**ImageSizeH** – Specifies the width in pixels of images stored.

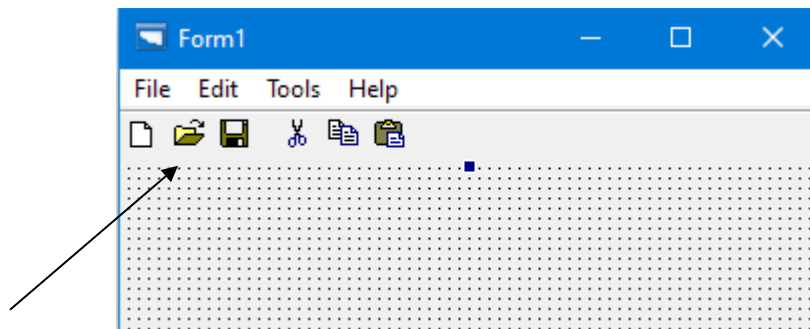
**ImageSizeV** - Specifies the height in pixels of images stored.

## **MORE STANDARD CONTROLS - continued**

### **ImageList and ToolBar - continued**

Toolbars are seen in almost all Windows applications. They provide the user a quick method of accessing commonly used functions. These functions should always be available elsewhere in the program and usually are menu selections.

Because the Toolbar control consists of images, adding a Toolbar can make your application more visually appealing. The Toolbar control can appear in different locations but is commonly placed at the top just below the menu.



ToolBar

The ToolBar is a graphical bar that consists of images that represent ToolButtons. These buttons are usually shortcuts to commonly used application functionality.

You must use an ImageList control to populate the ToolBar control with images. The ToolBar supports three distinct ImageLists – the normal imagelist, a disabled imagelist and a hot imagelist. The images reflect the state of each ToolButton.

The Tag Property will report the clicked ToolButton during execution. ToolTips are also normally provided for each ToolBar button.



## **MORE STANDARD CONTROLS - continued**

### **ImageList and ToolBar - continued**

Several styles of ToolButtons may be defined:

**Drop-Down** -A drop-down style button displays a list when the button is clicked.

**Drop-Down NW** – A button with a drop-down arrow but not as a separate section.

**Push** – A standard push button.

**Toggle** – A dual-state push button that toggles between the pressed and not pressed states each time the user clicks it. The button has a different background color when it is in the pressed state.

**ToggleGroup** – A button that stays pressed until another button in the group is pressed - similar to radio buttons.

**Separator** - A separator provides a small gap between button groups. A button that has this style does not receive user input.

### **ToolBar Properties**

**BtnHeight, BtnWidth** – Defines the size of the ToolButtons.

**ImageList, ImageListD, ImageListH** – Specifies the associated ImageList objects for normal, disabled, and hot states respectively.

**TextAlign** – Defines the Alignment of the specified text on each button.

### **ToolButton Properties**

**ButtonStyle** – Defines the type of button used.

**ImageIndex** – Associates an image within the ImageList with the button.

**Partial Push** – Allows the button to alter its appearance when pushed.

**Pushed** – Reports or sets the state of a push button.

**Tag** – Allows association of a user defined value with the button that is reported if using an Activate instruction. Normally the event result contains the zero-based button number.

## **EXERCISE – ADD A TOOLBAR TO THE INQUIRY FORM**

1. Add an imagelist to the inquiry form.
2. Load images into the ImageList using method calls or the Designer's imagelist editor.
3. Add a ToolBar to the form
4. Add a New, Save, Cut, Copy, Paste, Delete, and Find function.
5. Code simple routines for the functions defined to display an alert box indicating the button selected. We will add the actual code at a later time.

Customer Inquiry

File Edit Help

Customer Number:

Name:

Address:

City:  State:

Zip Code:

Contact Name:

Close

## MORE STANDARD CONTROLS - continued

### StatusBar



Status Bars display messages at the bottom of the form. They generally provide additional information such as page numbers, Keyboard status, file names, or the current date or time.

There are two kinds of status bars. Simple status bars display a single message. To create a simple StatusBar, ensure that the ShowPanels property is false and place your message in the Text property either at design time or during program execution.

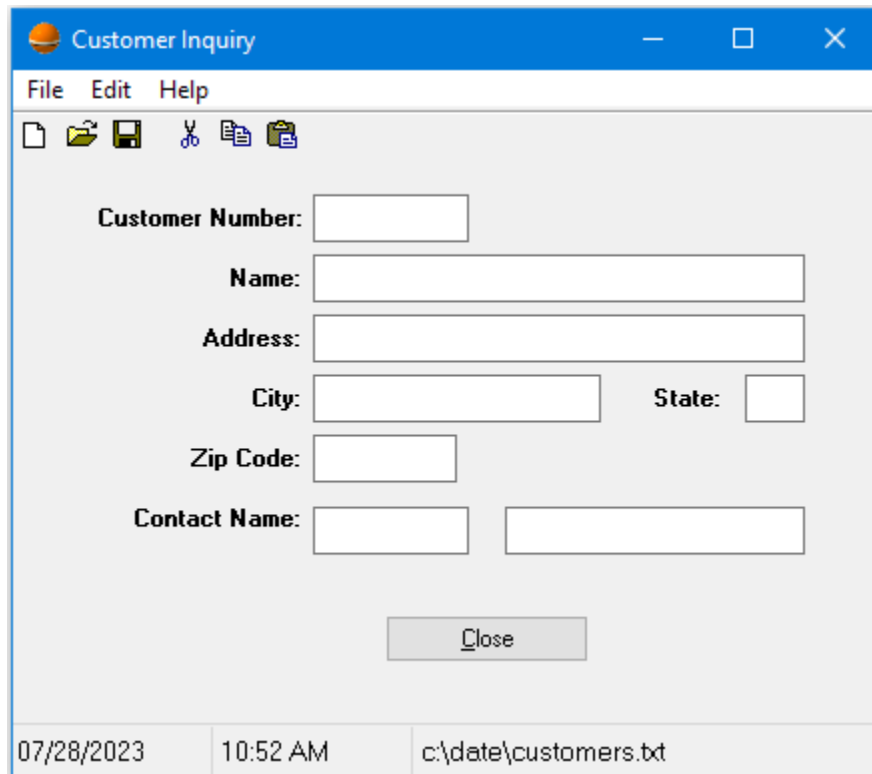
The screenshot shows a Windows-style application window titled "Customer Inquiry". It has a menu bar with "File", "Edit", and "Help". Below the menu bar is a toolbar with icons for file operations. The main area contains a form with the following fields: "Customer Number:" (single line), "Name:" (single line), "Address:" (single line), "City:" (single line), "State:" (single line), "Zip Code:" (single line), and "Contact Name:" (two single lines). A "Close" button is located at the bottom right of the form area. At the very bottom of the window, there is a status bar with the text "Customer inquiry in progress..." and a small blue square indicator.

**Simple Status Bar**

## MORE STANDARD CONTROLS - continued

### StatusBar – continued

A status bar with panels allows display multiple messages.



The screenshot shows a Windows-style application window titled "Customer Inquiry". It has a menu bar with "File", "Edit", and "Help". Below the menu bar is a toolbar with icons for file operations. The main area contains a form with the following fields: "Customer Number:" (a single-line text box), "Name:" (a single-line text box), "Address:" (a single-line text box), "City:" (a single-line text box), "State:" (a small single-line text box), "Zip Code:" (a single-line text box), and "Contact Name:" (two adjacent single-line text boxes). A "Close" button is located below the form. The status bar at the bottom is divided into three panels: the first panel shows "07/28/2023", the second panel shows "10:52 AM", and the third panel shows "c:\date\customers.txt".

**Status Bar with Panels**

### StatusBar Properties

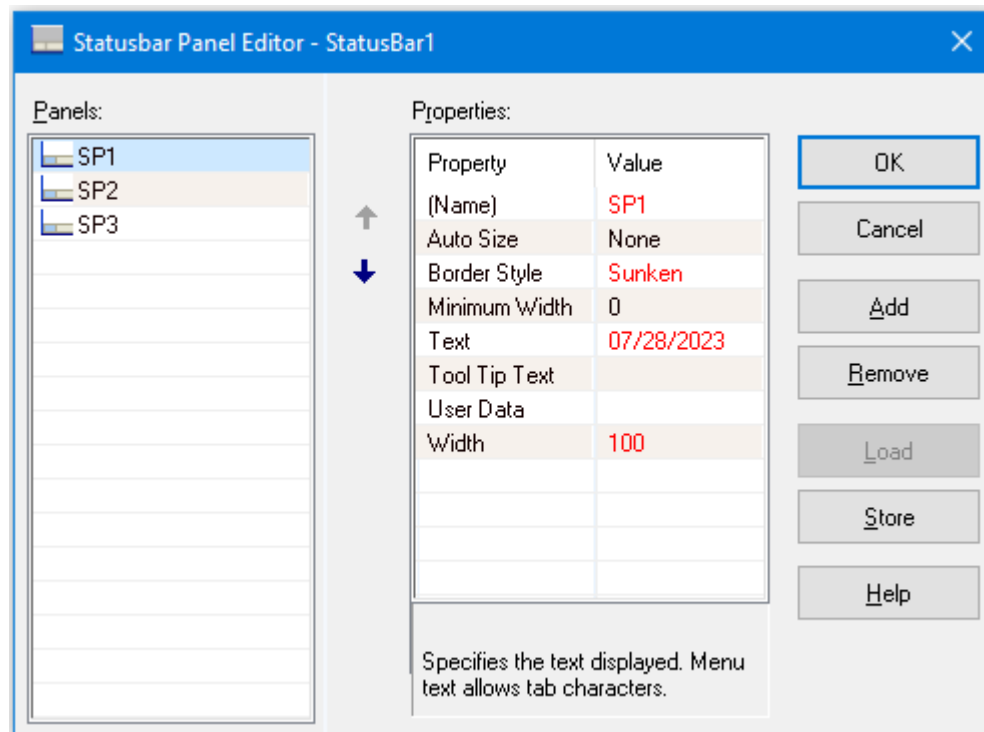
**ShowPanels** – Allows any defined panels to be displayed. Changes the StatusBar from Simple to Paneled.

**SizeGrip** – Displays a sizing handle in the lower right corner of the form.

## MORE STANDARD CONTROLS - continued

### StatusBar – continued

To add panels, you can use the AddPanel method at execution time or the StatusBar Panel Editor provided by the designer.



**StatusBar Panel Editor**

#### StatusPanel Properties

**AutoSize** – Allows automatic sizing of the panel based on the Text property value. Spring will expand the panel to consume any space not used by a panel.

**Minimum Width** – The minimum size of the panel.

**Text** – The data displayed within the panel.

**EXERCISE – ADD A STATUSBAR TO THE INQUIRY FORM**

1. Add an StatusBar with panels to the inquiry form.
2. Place the date in one panel and the time in a second.
3. Create a third panel that will be used for progress messages.
4. Add a timer to the form.
5. Code the timer event to update the date and time once every minute.
6. Experiment with the Minimum Width and AutoSize Spring properties.

## **MORE STANDARD CONTROLS - continued**

### **ListView**



The ListView control displays a list of items with icons. You can use a ListView to create a user interface like the right pane of File Explorer or the standard property window of the Designer.

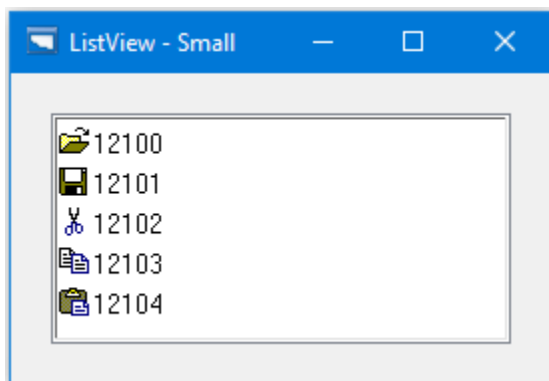
The control has four view modes that is controlled by the ViewStyle property: Icon, SmallIcon, List, and Report.

1. The Icon mode displays large icons next to the item text; the items appear in multiple columns if the control is large enough.
2. The Small Icon mode is the same except that it displays small icons.
3. The List mode displays small icons but is always in a single column.
4. The Report mode displays items in multiple columns.

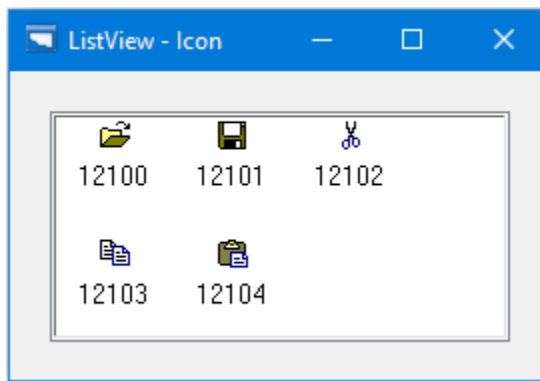
A listview only displays information. It does not have editing capabilities.

## MORE STANDARD CONTROLS - continued

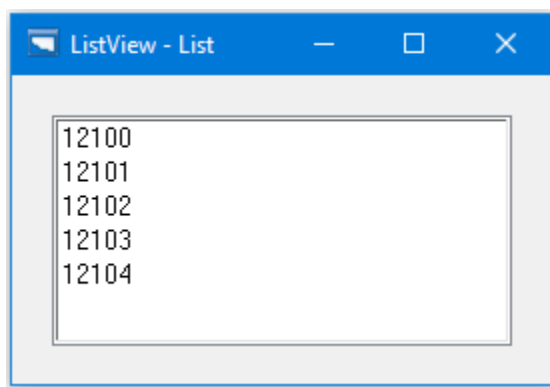
### Listview - continued



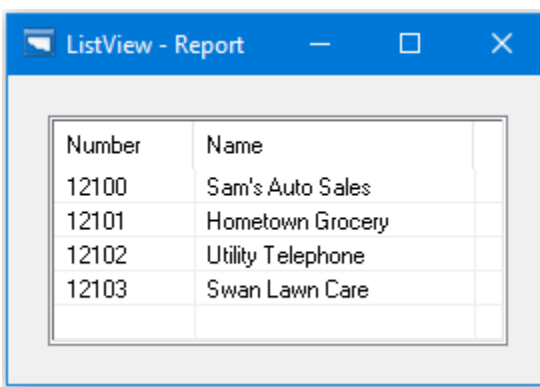
**ListView Small Icons**



**ListView Icons**



**ListView List**



**ListView Report**



## **MORE STANDARD CONTROLS - continued**

### **ListView – continued**

#### **ListView Properties**

**Activation** - Allows a generation of an ItemActivate event when an item is clicked or double clicked.

**Checkboxes** – Creates checkboxes for items.

**Full Row** - Controls the selection of rows within an object.

**Grid Lines** - Allows the display of grid lines around items and sub items

**Hide Column Headers** - controls the column header display while in Report view.

**Sort Header** - Indicates that column headers will act as buttons.

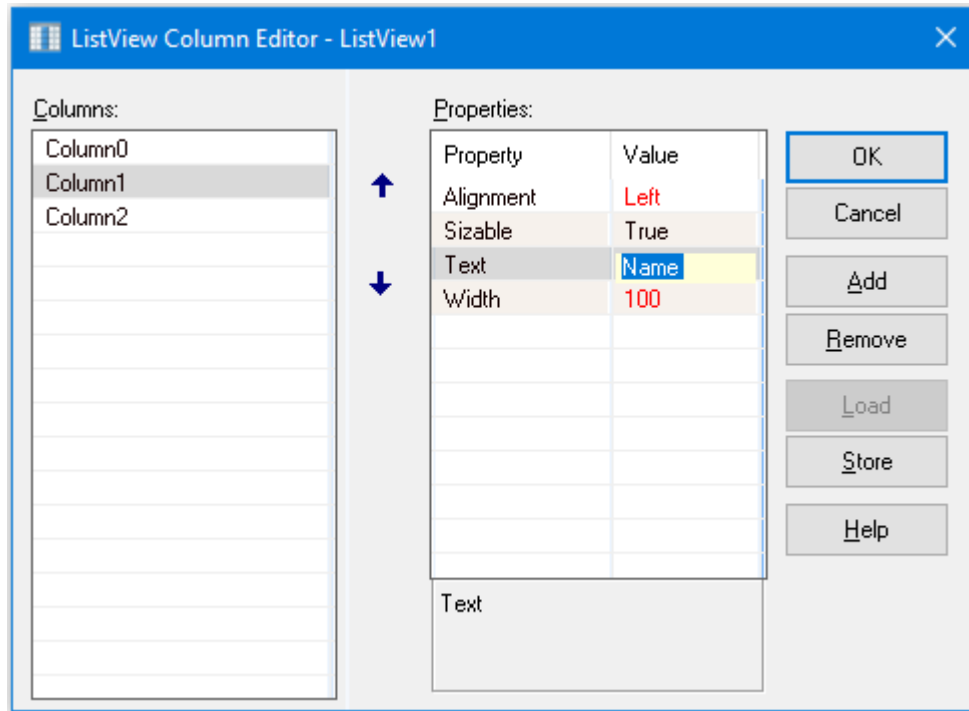
**Sort Order** – Orders the data based on the first column – ascending, descending, or none

**View Style** – Controls the format of the data displayed – List, Report, Small Icons, Icons

## MORE STANDARD CONTROLS - continued

### ListView – continued

A feature of the Designer is a ListView Column Editor. By selecting a ListView and clicking the ellipses on the “Columns” property, the following dialog is shown:



**ListView Column Editor**

This dialog allows quick entry and maintenance of ListView Columns. To add a column, click "Add" and then set the column's name and text in the Properties pane. It is suggested that a naming convention be followed just as with other controls.

Columns may be arranged within the ListView by selecting the column and then using the up and down arrows between the two panes.

Selecting a column and clicking the Remove button will delete the column.

The Store button allows the current column definitions to be saved for use in other forms. After clicking the button, simply provide a descriptive name. When editing another form, use the Load function to retrieve the saved definitions.

When all changes are complete, click OK to save the menu or Cancel to abandon the changes.

## **MORE STANDARD CONTROLS - continued**

### **ListView – continued**

#### **Color Support**

The LISTVIEW object allows specification of the foreground color, background color, and font attributes for individual cells using five object methods.

The **InsertAttrColumn** method adds a hidden column to the ListView that contains attribute data strings. The **SetItemText** method may then set a specialized formatted attribute string for a row item in the attribute data column. The attribute data string is formatted to provide default settings followed by individual column attribute settings that are used for a specific ListView row.

The {default} and {column} strings are composed of three (3) subfields that provide index numbers separated by a comma. The {default} and each {column} subfield group must be separated by a semicolon. The format of the attribute data strings must be given as follows:

```
"{default};{column1};{column2};...;{columnx}"
```

Where:

*default*

Optional. The {default} field gives default foreground color, background color, and font indexes for this row. The {default} field is composed of three subfield indexes that reference attribute tables that have been setup for a LISTVIEW object.

*column1 - columnx*

Optional. The {column1} field is optional and if specified gives the specific color and font attribute indexes for column one for this row. This same field definition format applies for all columns that exist in the LISTVIEW object.

## **MORE STANDARD CONTROLS - continued**

### **ListView – continued**

#### **Color Support - continued**

The string format for the {default} and {column} fields are defined as follows:

```
"{fgNDX},{bgNDX},{fontNdx}"
```

Where:

*fgNDX*

This subfield is a decimal number that is an index to retrieve a foreground color from the color table that has been created for the LISTVIEW object. The {fgNDX} decimal number value can be from one (1) to twenty (20).

*bgNDX*

This subfield is a decimal number that is an index to retrieve a background color from the color table that has been created for the LISTVIEW object. The {bgNDX} decimal number value can be from one to twenty.

*fontNDX*

This subfield is a decimal number that is an index into the font table that has been created for the LISTVIEW object. The {fontNDX} decimal number value can be from one to twenty.

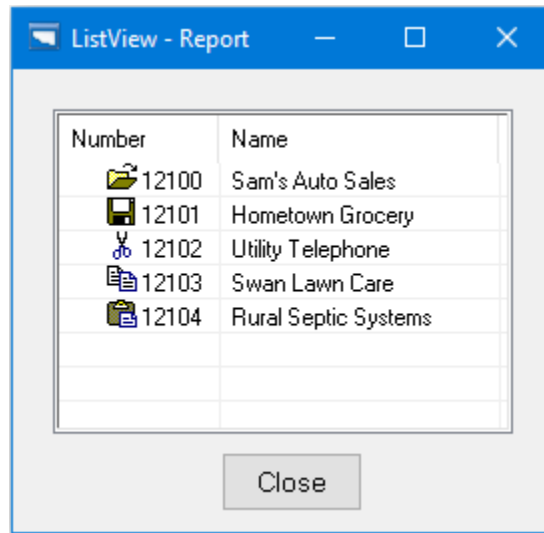
The general program logic flow to make use of the LISTVIEW color and font features to control individual cell attributes is:

1. Create a LISTVIEW object.
2. Use the InsertAttrColor, InsertAttrRGB, and InsertAttrFont methods to initialize the color and/or the font index tables for the LISTVIEW object.
3. Use the InsertAttrDefault method to set the default attribute settings.
4. Use the InsertAttrColumn method to put an attribute column into the LISTVIEW object.
5. Use the SetItemText method to define the attribute data strings for each row of the LISTVIEW desired. If an item in the attribute column does not have an attribute string, the default settings are used.

## Exercise – ListView

This exercise provides exposure to the ListView control:

1. Create a form containing a ListView and a button.



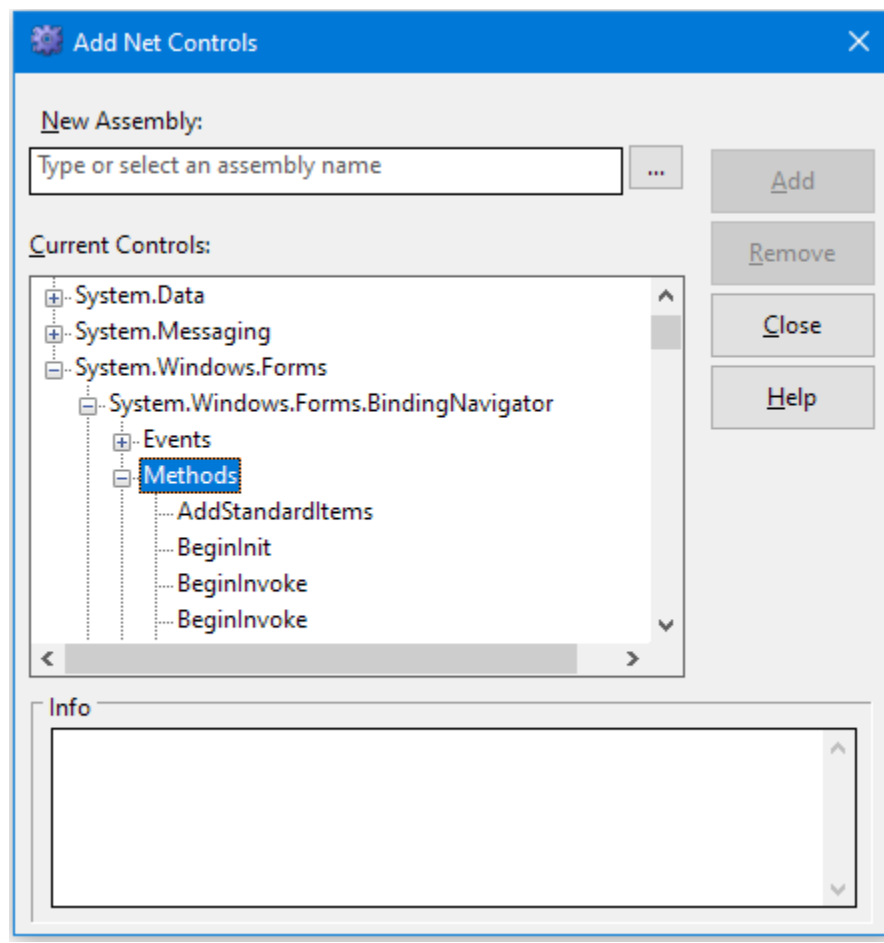
2. Allow sorting of the data by clicking on the column header.
3. Add color to the first column.

## MORE STANDARD CONTROLS - continued

### TreeView



The TreeView control is a hierarchical list of related node objects. The user can expand and collapse these node families easily while navigating through the control; depicting the trees and expanding and contracting the nodes are handled automatically by the TreeView control itself. An example of a TreeView control used in an application is the File Explorer program, where the TreeView occupies the left side of the form or the information show when examining a Net control object.

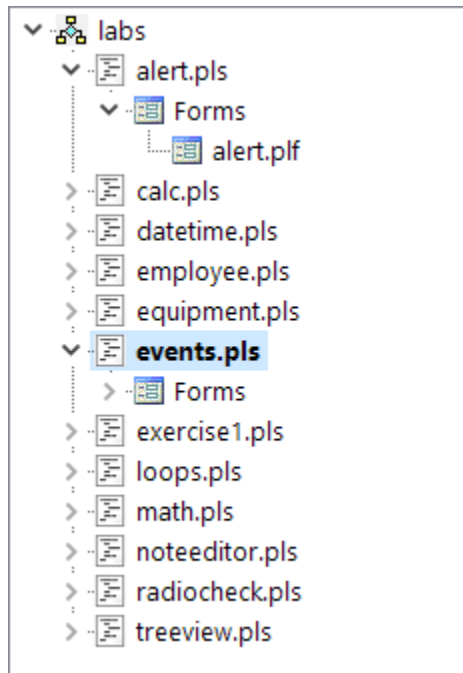


A form containing a TreeView

## MORE STANDARD CONTROLS - continued

### TreeView – continued

An ImageList object may be associated with the TreeView to give an enhanced representation of the data. The IDE uses this technique in its Source Map Window.



A Treeview with Icons



## **MORE STANDARD CONTROLS - continued**

### **TreeView – continued**

#### **TreeView Properties**

TreeViews support a number of properties that affect both the appearance and operation of the control.

**Checkboxes** – Add checkboxes to the left of each item in the control. This provides a method of selection.

**Full Row** – Allows clicking on any part of the row to select it.

**Hide Selection** – Hides the selected item when the object loses focus.

**Hot Tracking** – Underlines the selected items text as the mouse moves over it.

**TreeLine Style** - Indicates the type of lines drawn between parent and children items of a TREEVIEW object

**Use Buttons** - Controls the display of plus (+) and minus (-) buttons adjacent to parent items of a TREEVIEW object.

#### **TreeView Methods**

TreeViews support a number of methods that make their use simple and straight forward. One important item to understand is that each node of three has a handle. This numeric value is returned when nodes are added and employed when modifying or deleting nodes.

Within the Designer, each TreeView is shown containing sample data. That data is for design purposes only and is removed when the form is saved.

To fill a TreeView, begin by adding a root node to the tree. This is done with the InsertItem method as follows:

```
Tv.InsertItem GIVING RootHndl USING "Root Item",  
TVI_ROOT,TVI_LAST
```

The item is inserted using the description. The TVI\_ROOT parameter identifies the item as the root item. The TVI\_LAST parameter specifies the item should be added to the bottom of the tree. Other options include TVI\_FIRST for the first item, TVI\_SORT to add the item in sorted order or a zero-based item number.

## **MORE STANDARD CONTROLS - continued**

### **TreeView – continued**

#### **TreeView Methods - continued**

To add a child node to the root node, we again use the `InsertItem` method:

```
Tv.InsertItem GIVING ChildHndl USING "First Child":  
                RootHndl,TVI_FIRST
```

Note that this item was added using the handle of the root item. That specification makes the new item subordinate to the root. To add a child to the first child item use:

```
Tv.InsertItem USING "First Child",ChildHndl,TVI_FIRST
```

Of course many child nodes can be added at any level.

It is typically not necessary to attempt to save any of the handles unless their use is imminent. It is easy enough to retrieve the selected item's or the root handle using the `GetNextItem` method

```
Tv.GetNextItem GIVING Handle USING 0,TVGN_ROOT // root item
```

or

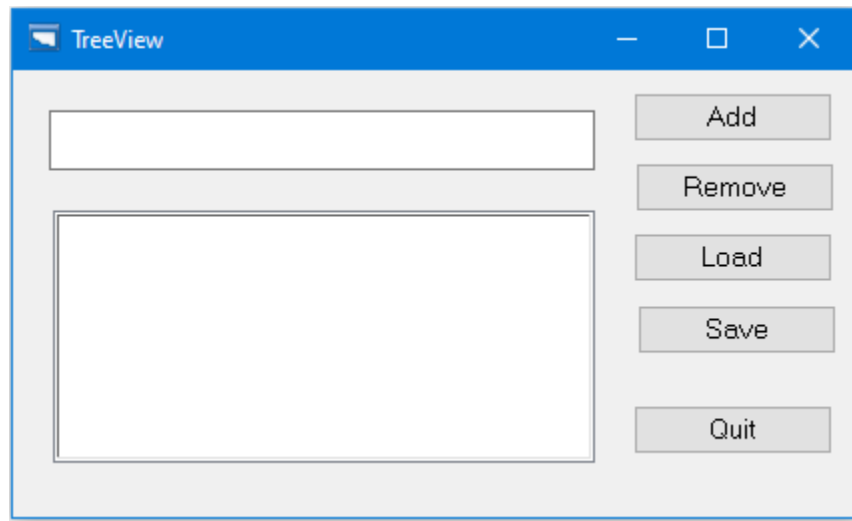
```
Tv.GetNextItem GIVING Handle USING 0,TVGN_CARET // selected item
```

The `GetNextItem` methods also provides the mechanism that allow you to step through all the entries at a particular level or navigate up or down within the tree.

The ability to load a `TreeView` directly using XML data is accomplished using the `LoadXMLFile` method. The `SaveXMLFile` method allows storing the data in a file.

## **EXERCISE – WORKING WITH TREEVIEWS**

1. Create a form that contains an EditText, a TreeView, and five Buttons as follows:



2. Code the Add button to add the text in the EditText object as a child to the selected item in the TreeView. Detect an empty TreeView and add the first item as the root item.
3. Code the Remove button to remove the selected item in the TreeView.
4. The Save button should output the data to an XML file.
5. The Load button should retrieve the saved data from the XML file.
6. Code the Quit button to halt the program.

## **ADDING FORMS TO A PROGRAM**

Most of the applications you create will have multiple forms. These applications may be either SDI (Single Document Interface) or MDI (Multiple Document Interface) style.

Windows Notepad and Wordpad are examples of an SDI application. Only one file is open at a time in a single editing window.

Microsoft Word and Excel are examples of MDI applications. Each has a parent program and child forms (documents) that are contained within the parent. You can create this type of application with Visual PL/B but most applications do not require it.

To add forms to your Visual PL/B application, choose "New Designer" from the Tools menu of the IDE. Once the Form Designer is displayed, click "New Form" from the File menu or press Ctrl+N. Alternatively, you may click the "New" button in the toolbar. A blank form will then be displayed.

Before populating the new form with objects, it is best to save the form and assign it a file name. Choose "Save" or "Save As" from the File menu or simply click the Save button in the toolbar, navigate to the correct directory, and specify the file name with a PLF extension.

## **ADDING FORMS TO A PROGRAM - continued**

After creating the new form, you should next assign the form a meaningful object name. Using the form designer, select the form and set its name and title properties. Forms typically use a “frm” prefix for their name.

When you have a single form application, you normally want to have that form visible at all times. However, when you are adding other forms to your application, you will want to control how and when the additional forms appear on the screen. Use the following properties to control the display of other forms:

**Visible Property** – This property controls the initial state of the form and all of its objects. For single form applications, this should always be set to True. For multiple form applications, this property should be set to False for all but the initial form. The property is then toggled using program code to make the form appear and disappear as it is needed. If the form was previously loaded using a FORMLOAD instruction, the following will make the form visible

```
SETPROP    frmAbout,VISIBLE=1
```

If you prefer to load and unload the form rather than toggling visibility, set the form's visible property to true and use a FORMLOAD at the point in the program where the form should be seen and a DESTROY when the form should be hidden. The disadvantage to loading and destroying the form is that the form objects are not available to code within the remainder of the program.

**Window Type Property** – This property controls the behavior of the form once displayed. Of particular importance is the distinction between a modal and a modeless form. Modal Dialog forms seize control when visible. The user is required to complete the modal form before working in other forms of the application. Code associated with the form is responsible for closing it and returning control to the parent form. Modeless Dialog forms allow the user to shift the focus from one form to another modeless form within the same program.

## **ADDING FORMS TO A PROGRAM - continued**

Once the form has been created, you need to associate it with the main program. Note that a single form may be included in any number of programs. To add the form to your application, return to the IDE and edit the main program source inserting a line similar to this one:

```
ABOUT      PLFORM      about.plf
```

The first "about" is a unique PL/B data label that identifies the form in the code. PLFORM is a PL/B instruction that includes the form in the program. The "about.plf" is the disk file name of the form to be added. You should substitute the correct file name.

This line should be inserted before any code that might reference the new form. For example, if you had a main program form named MAIN and it had a menu item that activated the ABOUT form, the source file should appear as:

```
ABOUT      PLFORM      about.plf  
MAIN        PLFORM      custinq.plf
```

After adding this statement, you will need to save the source file and choose "Refresh Dependencies" in the IDE shortcut menu to update the Source Map immediately.

## **ADDING FORMS TO A PROGRAM - continued**

Next, ensure that the form is loaded by the program. Normally, programmers load the form at the beginning of the program and simply toggle its visibility as it is required. To use this approach, make sure the form's visible property is set to False and add this statement to the beginning of the program source:

```
FORMLOAD      ABOUT
```

FORMLOAD is the PL/B instruction that places the form and its objects into memory. "ABOUT" is the label that is associated with the PLFORM definition.

Finally, to make the form available at the correct point in the program, it must be activated using an instruction such as:

```
ACTIVATE      ABOUT
```

Or

```
SETPROP       frmAbout,VISIBLE=1
```

After the form processing is complete and you want the form removed from the screen, you can close the form by using:

```
DEACTIVATE    ABOUT
```

Or

```
SETPROP       frmAbout,VISIBLE=0
```

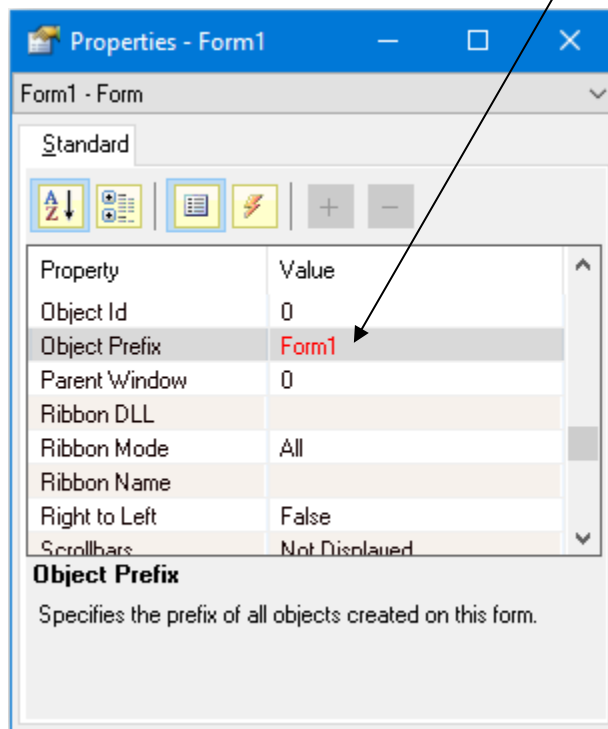
## **ADDING FORMS TO A PROGRAM - continued**

An item to note when using multiple forms in an application is that is quite easy to end up with duplicate object names. For example, if two forms each have a button name "cmdOK", the program will not compile. Object names must be unique throughout the application.

One solution to this problem is to create your own naming convention and then ensure that every item on each form follows that naming convention.

The Form Designer provides a second solution to this problem. Each form has a property named "Object Prefix". Additionally, an item in the Options dialog of the Form Designer is "Use Object Name Prefix". Specifying a unique Object Prefix value in each form and enabling the "Use Object Name Prefix" option will create all objects with a unique name.

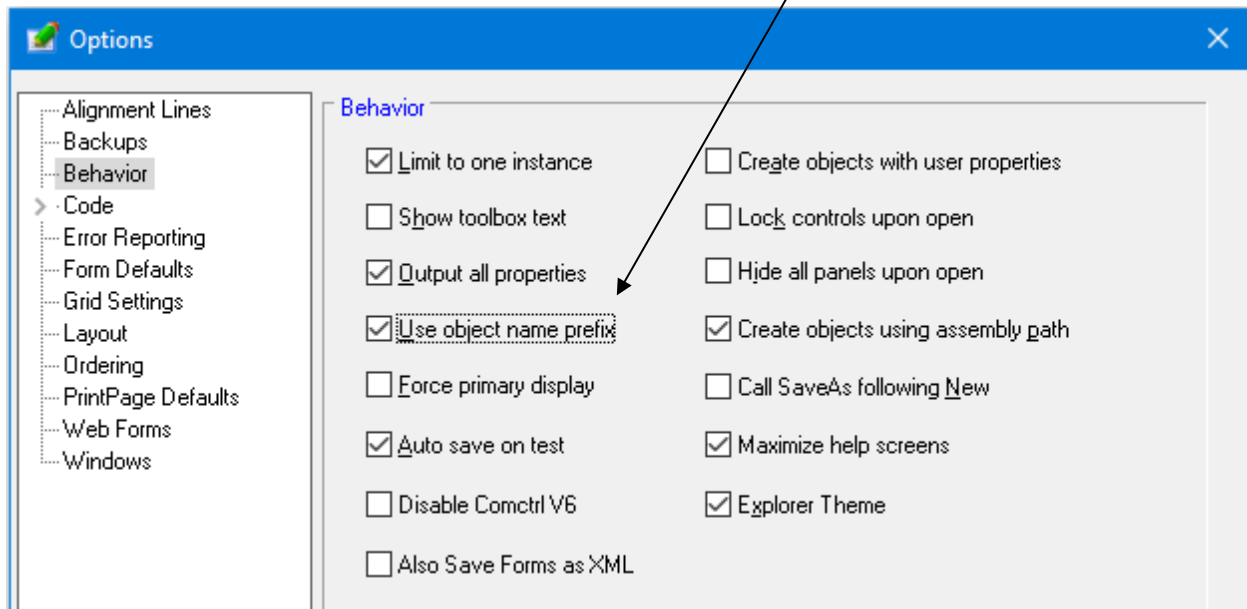
Object Prefix Property





## **ADDING FORMS TO A PROGRAM - continued**

Option to Enable Object Name Prefixes



Note that only new objects created will have the prefix applied. The designer does not rename any existing objects.

## **REMOVING FORMS FROM A PROGRAM**

You may need to remove a form from your application from time to time as program changes dictate. To remove a form:

1. Remove the PLFORM definition statement.
2. Remove the form ACTIVATE statement.
3. Remove any references to the form in the program source or in other forms.

**Note**

This removes the form from the program. It does not delete it from the directory in which it is stored if you had saved it previously. To delete a form that you are no longer using, you need to use the File Explorer. Make sure you have removed it from all Visual PL/B programs before you delete it from the directory.

**EXERCISE – ADDING FORMS**

1. Add a new blank form to the Customer Inquiry program.
2. Change the form name to "frmAbout" and give the form an appropriate title. Set the Visible property to false.
3. Draw a LabelText on the form. Give it a caption that describes your application.
4. Draw a command button on the form. Change its name to cmdOK and its title to "OK".
5. Code the click event for the command button to deactivate the frmAbout form.
6. Code the Help/About menu item to activate this form modally.
7. To test the application, compile and execute it. Click the About item on the Help menu to display the new form. Clicking OK on the new form should return the user to the original program.
8. This completes the exercise.

## **CHAPTER FIVE**

### ***PRINCIPALS OF GUI DESIGN***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter will discuss some principals of good graphical user interface (GUI) design.

Upon completion of this chapter, you will be able to:

- Identify the features of a good graphical user interface.
- Understand the reasoning behind graphical design decisions.
- Create more intuitive interfaces.

## **CREATING A USER INTERFACE**

Fundamentally, graphical user interface (GUI) design is the process of building an interface that can be used. This is such an obvious statement, but it is true. It does not matter how "pretty" the interface is. If the product is not useable by the end-user, the interface is useless.

Before you can begin creating a user interface you need to:

- Define the purpose of the application.
- Know your intended audience.
- If your application will be used by beginners, simplicity is always the key. A more advanced audience can handle a more sophisticated interface.

Remember that designing a GUI interface is an iterative process. Rarely is your first cut going to be the final product. It takes trial and error, allowing the user to test along the way and provide feedback.

Many of the guidelines for designing a Microsoft Windows user interface are based on IBM's Common User Access (CUA) guidelines. The use and acceptance of the CUA guidelines has been widespread.

The goal of these guidelines is to assist developers in designing GUIs that have a common "look and feel". The end-user benefits because they should not have to learn a brand-new way of using the application.

## **CREATING A USER INTERFACE - continued**

Before you can design a GUI, it would be helpful to examine a user interface for a common product (not software).

Let's begin by looking at the user interface for a door. The placement of the door handle and its shape gives the user visual clues about which way the door swings and what you must do to open it.

Imagine what would happen if you placed the door handle on the same side as the hinges. Candid Camera once tried this stunt and even though the door swung easily when pushed on the side opposite the hinges, nobody figured it out! People stood there, pushing, and pulling on the door handle to no avail! The stunt violates people's model of how a door should work.

This illustrates the simple point that the people who use your interface are going to make assumptions about the way it works. They may or may not be accurate. Your job as an interface designer is to ensure that the assumptions your users make accurately reflect what the software is going to do.

So, what makes graphical user interface design so difficult? There are several reasons:

1. Users only know what they want and need after they see it in action. They get a better feel for what they want after they see a working model or prototype of what they asked for.
2. The guidelines that govern GUI design are constantly evolving. As technology changes, so will the guidelines.
3. A good GUI changes the user's needs because it generally can offer more features with which the user is not familiar. Once they realize these features exist, their requirements may change. While this improves the interface, it makes the design more difficult.
4. It is easy to let your ego get in the way. As a designer, you must be willing to make modifications to the initial design in accordance with the user's reactions and needs.



## **CREATING A USER INTERFACE - continued**

### **Use Consistency in Your Interface Design**

Keeping these difficulties in mind, what can you do to help ensure the success of your user interface? On the following pages, we will discuss some design principles that you need to keep in mind when you are creating any type of user interface, not just graphical user interfaces.

**Make Your Interface Consistent.** Try to keep from confusing your user. Consistency is the trademark of good design.

Look at how Microsoft, for example, has used consistency throughout their software products. If you do not know specifically how to use one of their software products, when you start the application, you immediately know where "Help" is going to be. This is because Microsoft has used consistent menus and toolbars throughout many of their suites of software products. This gives them all the same "look and feel" which makes it easier for the user.

The more people use graphical environments, the more they expect consistency between applications. They expect them to behave the same and woe to the application that does not! Even if the application accomplishes great things, users will not continue to use it if it is not consistent with the way they are accustomed to doing their work.

Also, ensure consistency within your applications. Avoid having the same function key or command perform different tasks on different screens. This can be not only confusing but also dangerous!

## **CREATING A USER INTERFACE - continued**

### **Guide the User Through the Interface**

Consistency is important because you need to provide the same behavior in response to user actions. You must also provide clear and obvious paths for the user to accomplish their work.

A good interface gives clues about what to do next in each situation. Do not let the user become lost or confused. One of the strongest features of a graphical interface is the ability to provide good visual feedback to the user.

A good rule of thumb is to keep the interface from being too "cluttered". It can quickly become unclear to the user if there are too many options.

While it is important to direct the inexperienced user of your interface, you have to remember that a good interface will provide a way for the "power users" to accomplish more complex tasks without the interface getting in the way.

Designing menus, dialogs, and grouping options together logically play a large role in keeping your user from getting confused. For example, would you place a Print option inside the Edit menu? Hopefully not! Regardless of how good it looks or how well it performs, if it is confusing to use, people will not use it.

## **CREATING A USER INTERFACE - continued**

### **Make it User Friendly by Giving Good Feedback**

A good user interface must provide the right level and kinds of feedback in a timely manner.

An amusing anecdote describes two extremes of interface design: the yippee lap dog and the race car.

The yippee lap dog is constantly dashing around, yipping at every sound, licking your face, jumping in your lap, watching every move you make and generally being a nuisance. I know you have seen interfaces like this, too. You know the ones. "Are you sure you want to continue?" "Do you really want to save this file?" "You already have a file with that name. Do you want to replace it?" "Have you brushed your teeth today?" They demand confirmations and assurance before performing any tasks. This may be helpful at first for a new user but after some exposure, the users will start filtering all those messages - the trivial and the important.

The other extreme is the racing car. It is very powerful and just about anybody can drive it although not necessarily well. It is not going to ask you anything. It assumes you know how to drive it or you would not be behind the wheel. If you take it for a drive and put the pedal to the floor, it is not going to ask you if you really want to go 160 mph. Of course, if you go 160 mph and crash into a brick wall, it is not going to stop you from doing that either! It will let you do whatever you want and give you no warning – no feedback. These interfaces are as dangerous as the yippee lap dog is annoying.

The moral of the story is to try to reach a common ground between the two extremes. Feedback needs to be given to the user so that they know:

1. What they have done.
2. How they have done it.
3. What the results were.

## **CREATING A USER INTERFACE - continued**

### **Make it User Friendly by Giving Good Feedback - continued**

The user needs to know what they have done. When the user performs an action, they need confirmation. This information may be delivered in a passive way. For example, if the user chooses to print something, you need to display the print dialog, asking them for more information, such as the number of copies. If you simply send the document to the printer with no response or feedback, the user may get confused. Without feedback, they will often repeat the same command until they get some kind of response.

This is why the images on a toolbar often appear to "depress" when the user clicks on them with the mouse. This is a form of visual feedback. It confirms that the application's interface has recognized the users' action. The changed appearance of the controls on the screen is feedback to the user that an action has occurred.

The second type feedback goes together with consistency. The user needs to know how to do what they want. Following the print example above, be consistent with the commands in your interface. "Print" is used by most Windows applications. Do not try to call it something else such as "Output". The user may become confused by different terminology even though the result would be the same. If you always have "Print" as a File menu item or an icon on the toolbar, keep it that way because that is the way that the user knows how to print something from the application. A set of consistent actions should always produce the same results with the same feedback every time. Make it so they always know how to get the desired results.

## **CREATING A USER INTERFACE - continued**

### **Make it User Friendly by Giving Good Feedback – continued**

The third part of the feedback equation is to make sure that the user knows what the results were of their actions. In the print example, it is obvious that they should get something printed on the printer. If the action does not involve a piece of paper in their hands, maybe feedback on the screen is appropriate. If they delete something, give them confirmation of the deletion. If they add or change something, confirm that too. They need to see the results of their actions, or they may continue to unnecessarily repeat the steps necessary to perform the action!

For example, an application lets the user add a new record to the database. A user is looking at an existing record of John Smith. They issue the command to add a new record. However, after issuing the command to add a new record, they are still looking at that existing record of John Smith. This is confusing to the user of this product. Since they assumed they would see their newly added record and they did not see it, they assume that the record was not added correctly, so they repeat the steps to add the record. They then get a message that they have a duplicate and it will not add the record a second time! How exasperating! The solution is to simply code the interface so that they see the record that they just added – confirmation.

In summary, make sure you give timely and informative feedback. Do not question every move the user makes but be sure to tell them what they have done, how they have done it, and what the results of their actions were by some means of feedback - either messages or physical output from the application.

## **CREATING A USER INTERFACE - continued**

### **Human Factor**

Another important consideration in designing a good graphical user interface is the human factor. You need to understand how the eye is attracted to specific locations and or objects on the screen.

In general, you will need to understand four basic factors:

- 1. Motion**
- 2. Size**
- 3. Color**
- 4. Intensity**

### ***Motion***

The human eye is most sensitive to motion. Try changing your mouse pointer icon to a blinking arrow. When you first change the pointer, you will notice that your eye is attracted to the blinking arrow immediately.

If your application has motion, such as a video or some type of animation, a person will have a very difficult time trying to focus on any other area of the screen. For example, when you load software packages on your PC, many of them have a progress indicator of some sort. This serves two purposes. First, it lets the user know that the application really is working and secondly it gives your user something to watch so that they do not become bored or anxious.

It is important to limit your use of motion in the interface. Save it for important anomalies such as when your application is about to begin a process that could take some time to accomplish.

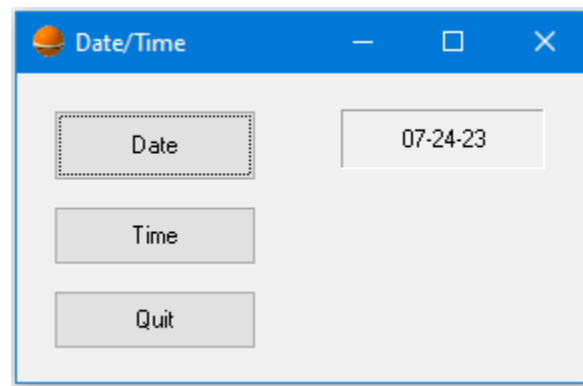
## CREATING A USER INTERFACE - continued

### Human Factor – continued

#### **Size**

The second most powerful stimulus to the eye is size. Did you ever set a full color graphic for your base window? Try plaid or fall leaves in Windows. You will find your eye is immediately drawn to the background instead of the objects on the screen.

Since the background of the screen is the largest screen component, stick with neutral backgrounds.



Consider the sizes of items on the screen as well. The larger sized items will attract the eye. Multiple large items intermixed with smaller items will be distracting. Notice that the objects on the screen above are all the same size. Therefore, your eye is not drawn to any one of them immediately.

## **CREATING A USER INTERFACE - continued**

### **Human Factor – continued**

#### ***Color***

Although color is at the lower end of the stimulus lists, it has many important considerations.

The number of colors on a single screen should be few and consistent. If there are too many colors, the eye will be confused and attracted to too many locations at the same time.

You may also want to consider color-blind users. You may even want to provide a color palette modification to all users so that they can change the colors in your application. Minimize the user of color for best acceptance.

It is appropriate to highlight values and relationships between values with like colors.

#### ***Intensity***

Intensity of color can play an important role in your design. Bright colors are attention getters. If you have a screen that contains multiple objects of the same size but one of them is bright red, that is where your eye will be drawn. In general, use high intensity for alarms, critical values, and possibly even status changes.



## **CHAPTER SIX**

### ***PL/B PROGRAMMING***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on the PL/B language elements and programming.

Upon completion of this chapter, you will be able to:

- Understand the steps involved in designing a program.
- Understand how variables are used in a Visual PL/B application.
- Learn how to use assignment statements and functions to handle string and numeric data.
- Define how constants are used.
- Identify the scope of a variable or constant based on how and where it is defined.
- Demonstrate the use of statements, expressions, operators, and program control structures to create and enhance applications.

## **DESIGNING A VISUAL PL/B PROGRAM**

To design a Visual PL/B program:

1. **Plan what you want the application to do.** What is the goal or mission?
2. **Plan how to solve the programming problem at hand.** What information (input) will the user enter into the program? How will the program process the information and what information (output) will the program produce?
3. **Design forms(s) that will accomplish the mission you set out to accomplish.** What will the program look like when it starts?
4. Finally, you need to **write code for the application** to make it perform the tasks required to solve the problem.

The Visual PL/B Language contains several hundred verbs but a few dozen will handle most of your programming tasks.

## **EXAMPLE OF DESIGNING A VISUAL PL/B PROGRAM**

Design the Investment Calculator Program.

### **What is the mission?**

You need to design an application that allows calculating of investment savings

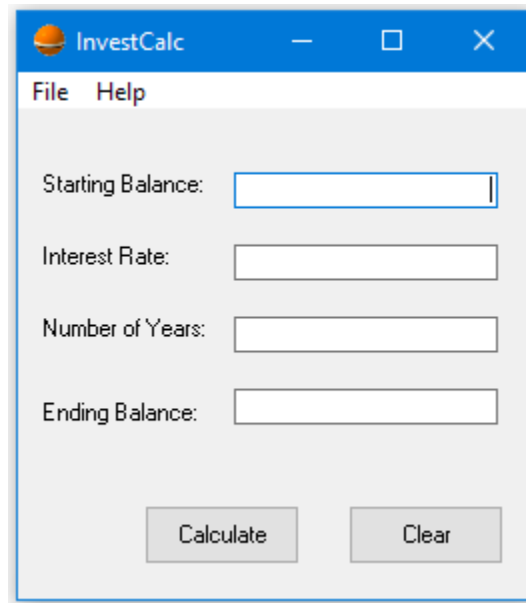
### **What information (input) will the user enter? How will the program process the information and what information (output) will the program produce?**

You want the user to enter information on a form such as a beginning balance, interest rate, and term. You will then collect the users' input values, validate them, and use a mathematical formula to calculate the ending balance. The ending balance will be displayed.

### **What will the program look like when it starts?**

You need some text boxes to allow user input, a command button they can pick to perform the tasks of collecting and validating the user input, performing the calculation, and displaying the result.

We will begin by designing the form. Next, we will explore the code needed to accomplish the task.

The image shows a screenshot of a Windows-style application window titled "InvestCalc". The window has a blue title bar with standard minimize, maximize, and close buttons. Below the title bar is a menu bar with "File" and "Help" options. The main area of the window is light gray and contains four labeled text input fields stacked vertically: "Starting Balance:", "Interest Rate:", "Number of Years:", and "Ending Balance:". At the bottom of the window, there are two buttons: "Calculate" and "Clear".

## **EXERCISE – THE INVESTMENT CALCULATOR**

1. Create a new file (calc.pls) within your project.
2. Add a File menu that contains a single item named Exit. Code the Exit item to end the program
3. Add a Help menu that contains a single item named About. Do not add any code to this menu at this time.
4. Design a new form with four StatText objects, four EditText objects, and one command button.
5. Set the properties of the form and controls as follows:

<b><u>Object</u></b>	<b><u>Property</u></b>	<b><u>New Value</u></b>
Form1	Title	Investment Calculator
	Name	frmInvest
Button1	Title	Calculate
	Default	True
	Name	cmdCalc
StatText1	Text	Starting Balance:
StatText2	Text	Interest Rate
StatText3	Text	Number of Years
StatText4	Text	Ending Balance
EditText1	Text	[blank]
	Name	txtStart
	Alignment	Right
	EditType	Decimal
EditText2	Text	[blank]
	Name	txtRate
	Alignment	Right
	EditType	Decimal
EditText3	Text	[blank]
	Name	txtYears
	Alignment	Right
	EditType	Decimal

**EXERCISE – THE INVESTMENT CALCULATOR – continued**

<b><u>Object</u></b>	<b><u>Property</u></b>	<b><u>New Value</u></b>
EditText4	Text	[blank]
	Name	txtEnd
	Alignment	Right
	EditType	Decimal

6. Save the form and test the program.
7. This completes the exercise.

## **CODING THE INVESTMENT CALCULATOR**

The next step is to look at what code we need to accomplish the following tasks:

1. The user will start the program and enter values into the first three text boxes named txtStart, txtRate, and txtYears. The values they enter can be captured programmatically by accessing the Text property for each EditText object.
2. You need to calculate the ending balance and display it in the txtEndBal EditText object.
3. As a good programming practice, you should add comments to your code so that it is self-documenting.

To retrieve the text properties and perform the calculation, we need to define some variables and use assignment statements to retrieve the values from the text boxes into the variables.



## VARIABLES

Visual PL/B holds specific information needed by your application in variables. Variables provide a temporary storage area for the values needed. Variables must be explicitly declared prior to their use.

- Variable names can be up to thirty-two characters long with only the first seven and last character significant.
- Variable names must contain only letters, numbers, the dollar sign (\$) or and underscore (\_).
- Variable names must be unique within the entire program.
- Local variable names in forms or inclusion files may begin with a pound sign (#) to force creation of a unique name within the file.
- Variable names are not case sensitive.

Character string variables are created with the general syntax:

```
Variable  DIM  Size
```

Where Variable is the name and Size is the maximum number of characters the variable can store. Variables defined with the DIM instruction are initialized to spaces and have a form pointer and logical length value of zero. This indicates a null variable.

Alternately, a character string variable may be defined as:

```
Variable  INIT "Hello"
```

This creates a five-character variable that is initialized to the string "Hello".

## **VARIABLES - continued**

Numeric variables are created with the general syntax:

```
Variable1 FORM 5      // from -9999 to 99999
```

or

```
Variable2 FORM 7.2    // from -999999.99 to 9999999.99
```

Alternatively, you can store integer values using the syntax

```
Variable  INTEGER      Size
```

Where Variable is the name and Size is a one, two, three, four, or eight. The table below list the numeric ranges of each of the integer types:

Size	Variable Range
1	0 to 255
2	0 to 65,535
3	0 to 16,777,216
4	0 to 4,294,967,296
8	0 to 18,446,744,073,709,551,616

Examples of Variable Definitions:

```
BeginAmt  INTEGER    1      // 8 bit integer value
```

```
LName     DIM        20     // 20 byte string variable
```

```
Balance   FORM       5.2    // Number with 5 decimal  
           //          and two digits
```

```
CoName     INIT      "Sunbelt Computer Systems" // 24 byte  
           //          initialized character string
```

Numeric variables are automatically initialized to zero.

## Variables - continued

### Scope of Variables

All variables defined in PL/B have global or public scope. That means they are available from the line of definition forward including any included source files or forms.

Unique variable names in included files or forms are generated by placing a pound sign or hash mark (#) before the variable name. When the compiler encounters such a variable name, the pound sign is pre-pended with the file's inclusion letter or letters.

Example:

```
X          DIM          1
           INCLUDE      Y.PLS
MAIN       PLFORM       Z.PLF
```

If the source file "Y.PLS" contained a line defining a variable as

```
#X          DIM          1
```

The actual variable name generated would be "A#X" since the first inclusion file is assigned a letter of "A". If the "Z.PLF" form included the same line, the variable generated there would be "B#X". Since it is illegal to define a variable containing a pound sign normally, you are assured of a unique variable name when using this method. The variable can then be referenced just as any other variable:

```
MOVE       "Y",#X      // locally defined variable
MOVE       "N",X        // globally defined variable
```

## STATEMENTS

Both the online help and the PL/B Language Reference can provide you with information you need when coding your procedures.

**Comment Instructions** begin with a period (.), asterisk (\*) or plus sign (+) in the first column. The entire line is then a comment.

**Line comments** are programmer notes that follow any required operands for an instruction. The final operand and any comments must be separated by at least one space. The compiler also allows the "//" character pair to denote the start of a comments section for those instructions where the comment could be taken as part of the instruction.

Example:

```
RESET      NAME, 30      comment
RESET      NAME, 30      // comment (explicit)
.
CALL MENU OF F5          // "of F5" is a comment
```

**Assignment Statements** set and retrieve properties and store data in variables using the SETITEM/GETITEM or SETPROP/GETPROP instructions. For example, gathering information from an EditText object is done like this:

```
Name DIM   30
.
GETITEM    edtName, 0, Name

or

GETPROP    edtName, Text=Name
```

To include quotes in a string, use the forcing character (#):

```
SETITEM    txtMsg, 0, "She said #"Welcome!#" "
```

The GETITEM and SETITEM instructions generally reference the Value of the control. GETPROP and SETPROP normally access Attributes or Properties of the control. The Language Reference contains detailed documentation regarding the use of the SETITEM and GETITEM instructions with the various types of controls.

## **WORKING WITH STRINGS AND NUMBERS**

When using assignment statements to either set or retrieve properties or values, you must be cognizant of the various data types with which you are working. For example, the text property of an EditText object is string data. This means it can be an alphanumeric value. When assigning a value to a string, you must use quotes:

```
SETITEM    Textbox1,0,"Sample string property"  
MOVE      "Sample string property",message
```

To retrieve a text property into a numeric variable, you must perform a two-step process:

1. The value is first retrieved into a string.
2. The string is then moved to a numeric variable.

```
Data      DIM      15  
StartBal   FORM     7.2  
.  
          GETITEM   txtStartBal,0,Data  
          MOVE      Data,StartBal
```

When setting a text property with a value from a numeric variable, the procedure is reversed.

```
Data      DIM      15  
StartBal   FORM     7.2  
.  
          MOVE      StartBal,Data  
          SETITEM   txtStartBal,0,Data
```

Check the documentation closely regarding the type of variable each control and property requires.

**EXERCISE – CODING THE INVESTMENT CALCULATOR**

1. Begin by declaring numeric variables that will store the starting balance (7.2), the interest rate (2.1), and the years (2.1). The fourth variable will store the calculated ending balance (7.2).
2. When the user clicks the Calculate button, your program should take the values from the three EditText objects and store them into the appropriate variables.
3. After the text values have been stored into their variables, you need to calculate the ending balance. That calculation looks like this:

Work                  FORM                  7.10

.

```
MOVE      (1 + IRate / 36500),Work
POWER     (Years * 365),Work
MULT      StartBal,Work,EndBal
```

4. Display the ending balance in the EditText object's text property formatting it for currency by using the EDIT verb.
5. As a good programming practice, you should add comments to your code so that it is self-documenting.
6. Save your form.
7. Test your program by entering valid values in the first three EditText objects and clicking "Calculate". We have not added any code yet to validate the values in the EditText objects but we will in a subsequent exercise.
8. This completes the exercise.

## **CONSTANTS**

If you find your program contains values that reappear repeatedly, it can greatly improve your code's readability by declaring these values as constants.

A constant is a meaningful name that takes the place of a number or string that does not change. Although a constant somewhat resembles a variable, you cannot modify a constant or assign a new value to it as you can a variable.

Visual PL/B offer three instructions that provide similar benefits:

<b>Instruction</b>	<b>Allows assignment of a data label to a</b>
CONST	numeric value.
DEFINE	string of one or more characters.
EQUATE	binary, decimal, hex, or octal value.

## **CONSTANTS – continued**

The Visual PL/B package contains four files that define constants that may be used within your application. These files are released in the "code" directory and all have file extension of "inc". The files are:

<b><u>File</u></b>	<b><u>Contains constants pertaining to</u></b>
PLBEQU.INC	properties, methods, and general purpose use.
PLBMETH.INC	object methods.
ADMIN.INC	the ADMIN class of instructions.
PLBAPI.INC	the WINAPI instructions.

To use these predefined constants, simply INCLUDE the file at the beginning of your source program.

```
INCLUDE PLBEQU.INC
.
ABOUT PLFORM about.plf
MAIN PLFORM custinq1.PLf
```



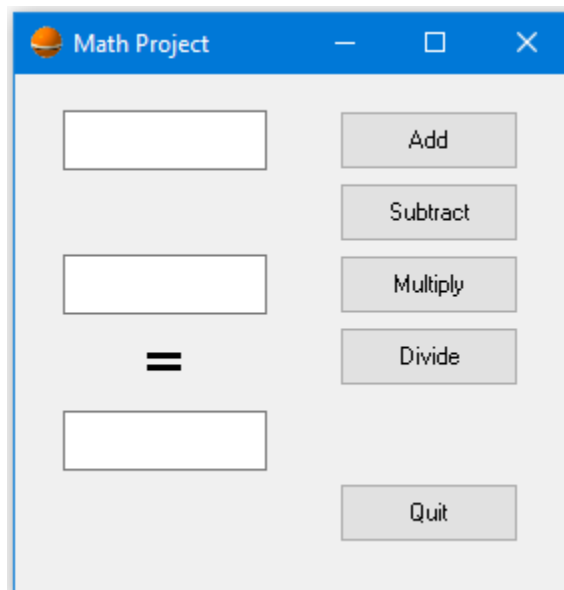
## EXERCISE – THE MATH PROJECT

The mission of this project is to design a simple calculator. It must be able to accept two numbers and add, subtract, multiply, or divide them. It must then display the answer.

1. Create a source file (math.pls) with the basic program code.

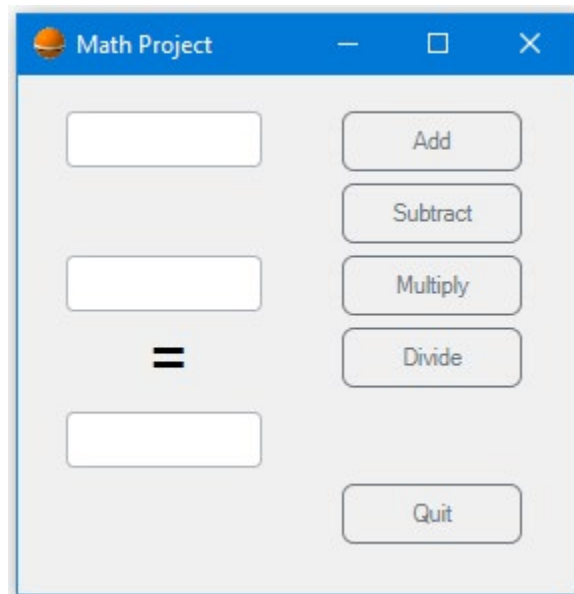
```
DATA          DIM   20
Value1        FORM  10
Value2        FORM  10
Value3        FORM  10
MAIN          PLFORM    MAIN.PLF
.
              FORMLOAD  MAIN
.
              LOOP
              EVENTWAIT
              REPEAT
```

2. Create a form (math.plf) with three EditText objects, two StatText objects, and five Command Buttons:



## **EXERCISE – THE MATH PROJECT - continued**

3. Remove the values in the text property of all three EditText objects and the first StatText object.
4. Place an equal sign (=) as the text for the second StatText object.
5. Set the name and title properties of the form, EditText, and Button objects as appropriate.
6. The first two EditText objects should only allow numeric input.
7. The third EditText object should not allow input.
8. Code the buttons to add, subtract, multiply, and divide the values in the first two EditText objects. Also, display the appropriate symbol in the first StatText object for the math operation you are performing.
9. You may want to change the font properties and alignment for the StatText objects.
10. Code the last button to terminate the application.
11. Save the form and test the application.



## **CHAPTER SEVEN**

### ***USING THE DEBUGGING TOOLS***

**CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on how to use the debugging tools in Visual PL/B to interact with the graphical user interface (GUI) during testing.

Upon completion of this chapter, you will be able to:

- Identify the three types of errors that can be encountered when testing a Visual PL/B application.
- Demonstrate how to correct logic errors using the debugging facilities available in Visual PL/B.

## INTRODUCTION TO DEBUGGING

There are three (3) types of errors encountered when programming:

1. **Syntax Errors** are errors caused by incorrectly structured source statements. These errors appear during the build process and must be corrected before the compiler generates any code.
2. **Runtime Errors** are unexpected results such as a file open failure or a file format error. Visual PL/B allows you to trap these errors and deal with them programmatically.
3. **Logic Errors** are issues that you can find and fix using the Visual PL/B debugging tools. These errors are not syntax or runtime errors but just the program not performing to our expectations.

**INTRODUCTION TO DEBUGGING – continued**

Some basic steps you can take to avoid errors are:

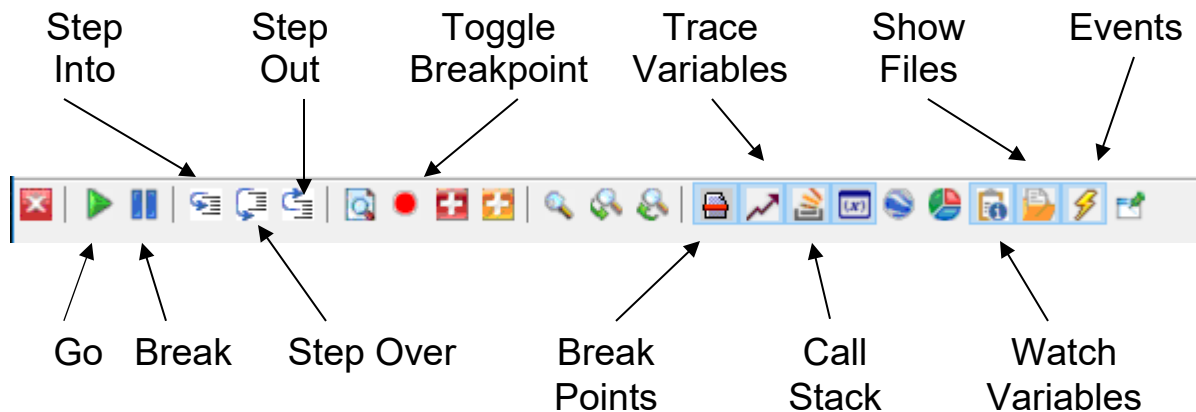
1. Design your application carefully. More design time means less debugging time.
2. Use comments and white space in your code to improve readability and remind yourself of what the purpose of the code is.
3. Use consistent naming conventions for your variables, objects, and program labels.

To test you application, you need to understand where you are in Visual PL/B:

- In design mode, you cannot execute code. You must build and run the program by pressing the F5 key or clicking the Build icon in the toolbar.
- In run mode, you cannot change source code. Terminate the program to return to design mode.
- In debug mode, execution is halted while running the application in the interpreter. You can view and edit variable contents, set breakpoints or trace points, examine subroutine calls, and step through your application. You cannot change source code.

## DEBUGGING TOOLS

The debugging tools available in Visual PL/B are accessed through the Debug menu and tool bar.



- **Step Into** executes the next line of code in the application. If the current line is a CALL statement, the next line displayed is the first line in the called routine.
- **Step Over** allows you to single step through the code but it will not follow execution into CALLED routines. The routines will still be executed.
- **Step Out** allows you to finish executing the current called routine and stop at the next line following the statement following the CALL.
- **Go** begins program execution stopping only when a breakpoint, DEBUG, or STOP statement is encountered or when an untrapped error occurs.
- **Break** halts the program execution at the next statement.
- **Toggle Breakpoint** will set or clear a breakpoint on the selected line.

**DEBUGGING TOOLS - continued**

- **Trace Variables** selects the panel that displays the variables that have trace points set.
- **Watch Variables** selects the panel that displays the contents and state of selected variables.
- **Breakpoints** identifies lines where execution will pause allowing the user to investigate the program's state and variables.
- **Show Files** display the status and information for all defined files.
- **Call Stack** shows the current state of the call stack.
- **Events** display the current event stack.



**DEBUGGING TECHNIQUES**

Breakpoints are set in debug mode. Program execution will be halted prior to executing the line of code that contains the breakpoint. The STOP instruction will also cause a break in execution as will the DEBUG instruction.

To interact with the program and use the debugging tools, you must:

1. Build the program.
2. Start the program for debugging.
3. Visual PL/B automatically enters debug mode when run using the IDE and one of the following events occurs:
  - At the first executable statement when running with debug
  - An untrapped run time error occurs when running with debug.
  - A DEBUG instruction is encountered.
  - Execution reaches a line on which a breakpoint is set.
  - Execution encounters a STOP statement.
  - A trace point expression becomes true.

**DEBUGGING TECHNIQUES – continued**

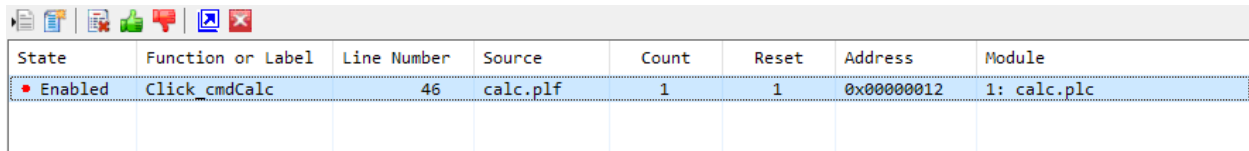
Typically, you set a breakpoint in the GUI debugger by placing the cursor on the line of code in the debug window at which you want the debugger to stop and select "Set Breakpoint" from the shortcut menu. The shortcut menu is displayed by clicking the right mouse button. An alternate method of selecting the line is to click in the white space at the beginning of the line.

Once you are in debug mode, you can interact with the program to:

- Determine which routines have been called.
- Control the statement executed next.
- Watch the values of variables and their attributes.
- Change the values of variables.
- Manually control the flow of statements in the application.

## DEBUGGING TECHNIQUES – continued

Breakpoints may also be managed through the use of the Breakpoint Panel



State	Function or Label	Line Number	Source	Count	Reset	Address	Module
• Enabled	Click cmdCalc	46	calc.plf	1	1	0x00000012	1: calc.plc

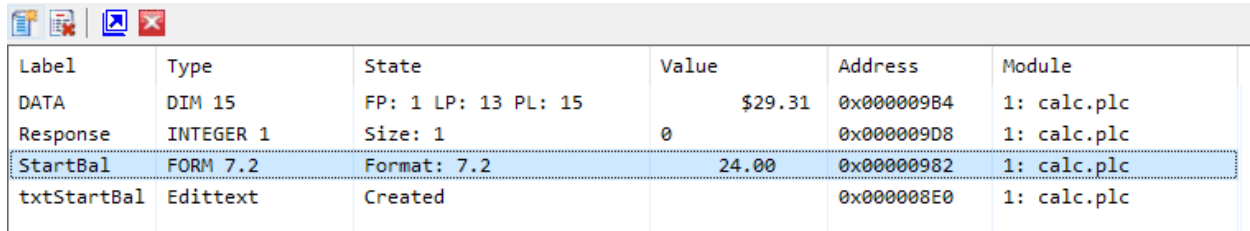
### Breakpoint Panel

The Breakpoint window lists all currently set breakpoints. It also allows breakpoints to be set, enabled, disabled, or cleared.

- **State** reports whether the breakpoint is enabled or disabled.
- **Function or Label** shows the Function name or closest execution label.
- **Line Number** is the line number in the source file.
- **Source** is the name of the source file.
- **Address** shows the label or program address of the breakpoint.
- **Count** is the number of times the breakpoint line is executed before entering the debugger.
- **Reset** is the value assigned to Count after entering the debugger.
- **Address** shows the label or program address of the breakpoint.
- **Module** shows the source file name that contains the breakpoint.

**DEBUGGING TECHNIQUES – continued**

You can view the value of variables through the use of the Watch Panel.



Label	Type	State	Value	Address	Module
DATA	DIM 15	FP: 1 LP: 13 PL: 15	\$29.31	0x00000984	1: calc.plc
Response	INTEGER 1	Size: 1	0	0x000009D8	1: calc.plc
StartBal	FORM 7.2	Format: 7.2	24.00	0x00000982	1: calc.plc
txtStartBal	Edittext	Created		0x000008E0	1: calc.plc

**Watch Panel**

The Watch panel lists all variables currently being monitored. It also allows for the addition and removal of watch variables. Information about each monitored variable include:

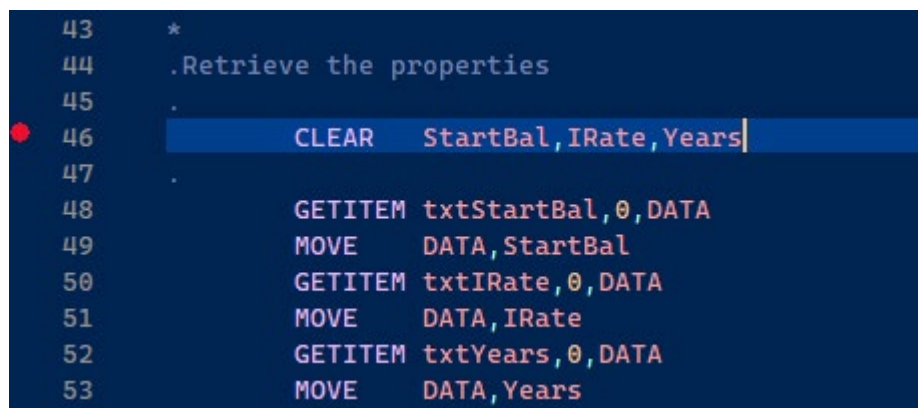
- **Label** is the variable name.
- **Type** is the PL/B data type of the variable.
- **State** shows type specific information. A DIM variable would have:
  - **PL** shows the variable's Physical Length (strings only).
  - **FP** displays the variable's current Form Pointer value (strings only).
  - **LP** is the value of the variable's Length Pointer (strings only).
- **Value** represents the current contents of the variable. To modify the value, double-click in the value column of the appropriate variable and enter a new value.
- **Address** shows the label or program address of the variable.
- **Module** shows the source file name that contains the variable.

## DEBUGGING TECHNIQUES – continued

We can explore the debugging tools and techniques using the Investment Calculator created in a previous exercise.

Select the program in the IDE project window and click "Build and Debug" from the toolbar. The program will be compiled if necessary and the debugger started.

Using the cursor keys scroll the source window until the code for the "Click\_cmdCalc" routine is shown. Set a breakpoint on the first executable statement in the routine by right clicking on the line and selecting "Toggle Breakpoint".



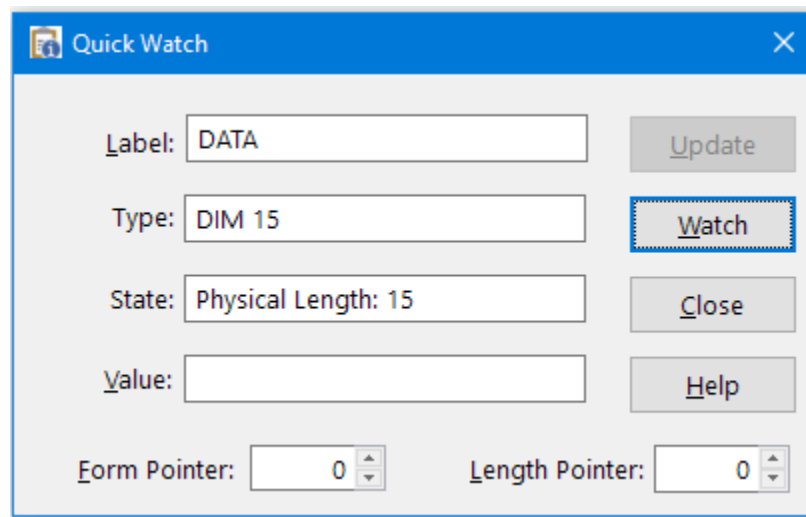
```
43      *
44      .Retrieve the properties
45      .
46      CLEAR    StartBal, IRate, Years
47      .
48      GETITEM  txtStartBal, 0, DATA
49      MOVE     DATA, StartBal
50      GETITEM  txtIRate, 0, DATA
51      MOVE     DATA, IRate
52      GETITEM  txtYears, 0, DATA
53      MOVE     DATA, Years
```

Press F5 or select Go from the toolbar to resume program execution.

Switch to the application window and enter values in the text boxes. Press Calculate to trigger the procedure with the breakpoint set.

## **DEBUGGING TECHNIQUES – continued**

Switch back to the debugger and set a Watch on the variable that receives the starting balance amount (DATA). This is done by right clicking on the variable and selecting "Quick Watch". The Quick Watch window will be displayed and the variable can be added to the Watch panel using the Watch button.

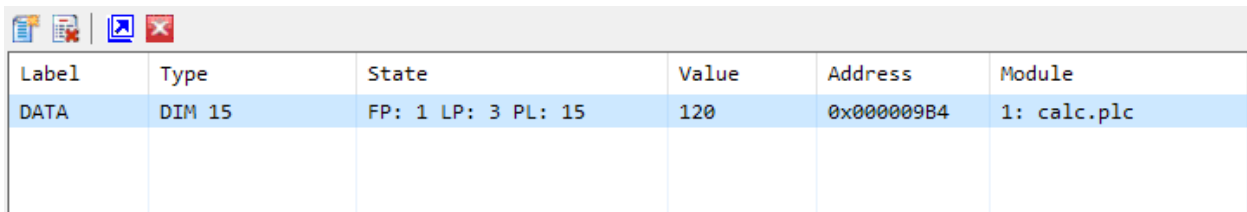


Note that the variable is currently null (has a Form Pointer of zero) and the value field shows blanks.

Next, we are going to single step through the routine. Press F8 or click "Step Into" on the tool bar two times. The yellow line indicating the next statement to be executed should advance one line in the debug window.

**DEBUGGING TECHNIQUES – continued**

Note that the contents of the Watch panel have changed to reflect the value placed into the variable by the GETITEM instruction. The Form Pointer, Length Pointer, and Value fields have also been updated to reflect the variable's state.



Label	Type	State	Value	Address	Module
DATA	DIM 15	FP: 1 LP: 3 PL: 15	120	0x000009B4	1: calc.plc

Should you wish to change the value of the variable, double click in the Value column of the variable. You may then enter a new value. The value is updated when the focus is changed to another part of the screen.

Once the value is changed, you may single step through the event routine a line at a time using the F8 key or press the F5 key to resume program execution.

Upon exiting the debugger, the program is terminated, and control is returned to the IDE.

**Note**

In the above example, we changed the value that had been retrieved from the first text box. That value was then used in the computation. We did not alter the contents of the EditText object and this causes the data in the form to appear incorrect.

**DEBUGGING TECHNIQUES – continued**

Step Over is identical to Step Into except when the statement executed is a CALL instruction. Step Over will execute the called routine as a unit instead of single stepping through it. When the routine returns, you will resume stepping in the current routine. This is helpful if you are certain the error you are searching for is not in the called routine. It speeds up the execution to ignore single stepping through called routines.

Step Out bypasses single stepping through the remainder of the code in a called routine. The remainder of code in the routine is executed and debugging will resume after the RETURN instruction.

Another convenient feature of the debugger is the "Run to Cursor Line" function. Simply select a statement in the code window and choose this function from the main or shortcut menu. The application will run until it reaches the line you have selected and then enter the debugger. The function acts as a soft breakpoint since the debugger is only invoked once for the selected statement and then it is automatically cleared.

With the "Set IP to Cursor Line" shortcut menu command, you can alter the program's flow without the need of exiting the debugger, modifying the source, and recompiling the programs. You can also use this function to repeat a series of instructions. This functionality combined with the ability to modify variables allows you to try program corrections without the need to recompile.



**EXERCISE – DEBUGGING**

Use the Investment Calculator or your Math Project to try out the debugging tools.  
Practice the following:

- Setting breakpoints
- Watches
- Stepping through code
- Modifying variable values
- Altering program flow

## **CHAPTER EIGHT**

### ***DATA ENTRY VALIDATION***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on additional Visual PL/B language elements and programming fundamentals as they apply to GUI programs.

Upon completion of this chapter, you will be able to:

- Demonstrate techniques for validating data entry using the ALERT instruction to interact with the user.
- Demonstrate how to set the tab order for controls on the form by using the TabID property.
- Create About dialogs using the Alter instruction.
- Use the CauseValidation property and the Validate event for data validation.
- Learn how to highlight text in an EditText at run time.
- Demonstrate how to prevent validation from occurring in certain situations.
- Learn how to use the EditDateTime and EditNumber controls to assist in validating data.

## THE ALERT INSTRUCTION

Visual PL/B provides a very useful instruction for interacting with your user. That instruction is ALERT.

ALERT allows you to display information to your user in a new window and return their response. This instruction creates a window complete with a title, an icon, your message, and the buttons you specify.

The syntax for the ALERT instruction is;

```
ALERT      [{Window};] {Type}, {Message} :  
           {Result} [, {Title}]
```

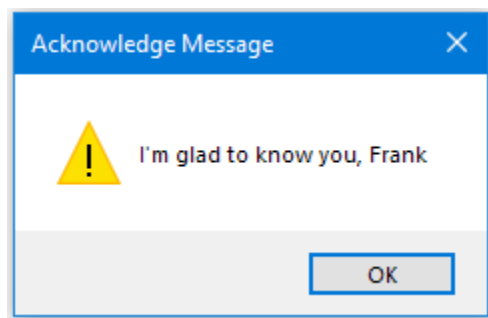
The parameters of the ALERT instruction are:

- **Window** is an optional WINDOW object variable or Form on which the ALERT is displayed. If this value is omitted, the ALERT dialog is displayed on the current window or Form.
- **Type** indicates the icon and the buttons that will be displayed in the dialog. This parameter is required.
- **Message** is the text you want displayed in the box. This parameter is required.
- **Result** is the variable that returns the user's response. This parameter is required.
- **Title** is the optional text displayed in the title bar of the ALERT window.

## THE ALERT INSTRUCTION – continued

As an example of an ALERT dialog, the instruction below generates the following dialog.

```
ALERT      CAUTION,"I'm glad to know you, Frank":
           Result,"Acknowledge Message"
```



**Example Alert**

There are several different types of ALERT formats available. The Type parameter defines the icon that is displayed and the number and names of buttons in the dialog. Both the icons and the buttons are controlled by the Type parameter. Visual PL/B offers four predefined types and gives you the ability to create your own combinations of icon and buttons.

The predefined ALERT types are:

Type	Displays an icon with ...
CAUTION	an exclamation mark and one button named OK.
NOTE	an "i" character and one button named OK.
PLAIN	a question mark and three buttons named YES, NO and CANCEL.
STOP	a stop sign and one button named OK.

## **THE ALERT INSTRUCTION – continued**

You may also create your own type of ALERT dialog by using a numeric variable for the Type parameter that contains a sum of the following values. The constants listed below are defined in PLBEQU.INC and are enabled by defining PLBALERT as a one (1) prior to the inclusion of the file.

<b>Value</b>	<b>Constant</b>	<b>Meaning ...</b>
0x000000	\$MB_OK	OK button. The message box contains one push button: OK.
0x000001	\$MB_OKCANCEL	OK/CANCEL buttons. The message box contains two push buttons: OK and Cancel.
0x000002	\$MB_ABORTRETRYIGNORE	ABORT/RETRY/IGNORE buttons. The message box contains three push buttons: ABORT, RETRY and IGNORE.
0x000003	\$MB_YESNOCANCEL	YES/NO/CANCEL buttons. The message box contains three push buttons: Yes, No and Cancel.
0x000004	\$MB_YESNO	YES/NO buttons. The message box contains two push buttons: Yes and No.
0x000005	\$MB_RETRYCANCEL	RETRY/CANCEL buttons. The message box contains two push buttons: Retry and Cancel.
0x000010	\$MB_ICONSTOP	Stop-sign icon. A stop-sign icon appears in the message box.

**THE ALERT INSTRUCTION – continued**

0x000020	\$MB_QUESTION	Question-mark icon. A question-mark icon appears in the message box.
0x000030	\$MB_ICONEXCLAMATION	Exclamation-point icon. An exclamation-point icon appears in the message box.
0x000040	\$MB_ICONINFORMATION	Information icon. An icon consisting of a lowercase letter 'i' in a circle appears in the message box.
0x000000	\$MB_DEFBUTTON1	Default button one. The first button is the default button. Note that the first button is always the default unless MB_DEFBUTTON2, MB_DEFBUTTON3 or MB_DEFBUTTON4 is specified.
0x000100	\$MB_DEFBUTTON2	Default button two. The second button is a default button.
0x000200	\$MB_DEFBUTTON3	Default button three. The third button is a default button.
0x000300	\$MB_DEFBUTTON4	Default button four. The fourth button is a default button.
0x010000	\$MB_SETFOREGROUND	Set foreground window. The message box becomes the foreground window. Internally, Windows calls the 'SetForegroundWindow' API function for the message box.

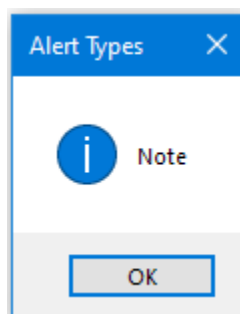


## THE ALERT INSTRUCTION – continued

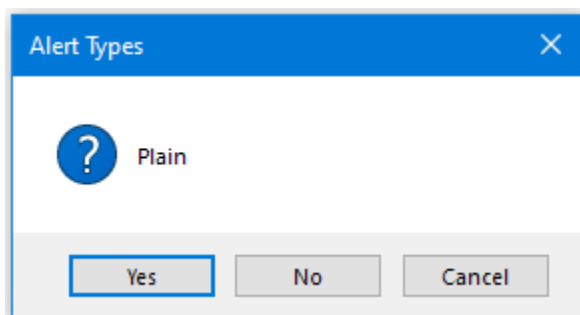
Samples of the various types of ALERT dialogs follow:



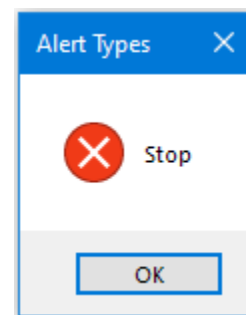
**Caution**



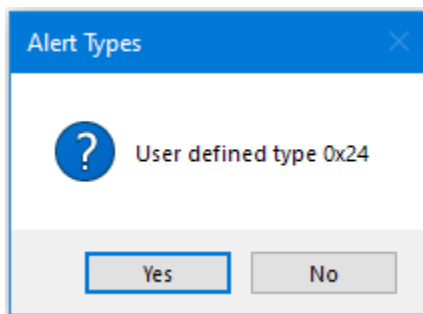
**Note**



**Plain**



**Stop**



**User Defined 0x24**

## **THE ALERT INSTRUCTION – continued**

ALERT boxes with a single button always return a value of one (1).

PLAIN ALERT boxes return a value indicating the user's choice. Possible return values are:

<b>Value</b>	<b>The button selected by the user was ...</b>
1	YES
2	NO
3	CANCEL

For user-defined ALERT box types with more than one button, the Result parameter signifies the user's choice. Possible return values are:

<b>Value</b>	<b>The button selected by the user was ...</b>
1	OK
2	CANCEL
3	ABORT
4	RETRY
5	IGNORE
6	YES
7	NO

## THE ALERT INSTRUCTION – continued

ALERT Instruction Example:

Click\_cmdTest

```
ANSWER      INTEGER      1
ATYPE       INTEGER      1, "0x24"

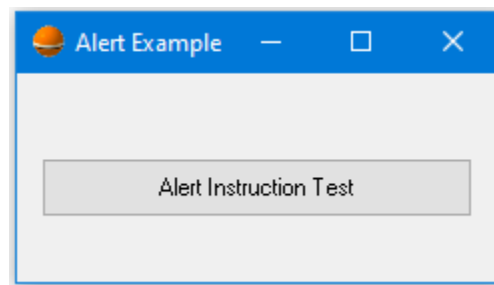
            ALERT        TYPE=ATYPE, "Are you having fun?":
                               ANSWER, "Question"

.
            IF           (ANSWER = 6)
            ALERT        NOTE, "I'm glad!", ANSWER
            ELSE
            ALERT        CAUTION, "We'll keep trying!", ANSWER
            ENDIF

.
            RETURN
```

## **EXERCISE – THE ALERT INSTRUCTION**

1. Start a new application within your project.
2. Add a form and give it a title.
3. Draw one button on the form and name it "cmdTest". You may change the title if you wish.
4. Place the sample code from the previous page in the click event for the button.
5. Modify the Alert type to include a "Cancel" button.
6. Terminate the application if the Cancel button is clicked.
7. Run the application to test the code. You may want to set a breakpoint at the beginning of the code so that you can watch each line of code execute.



**Example Form**

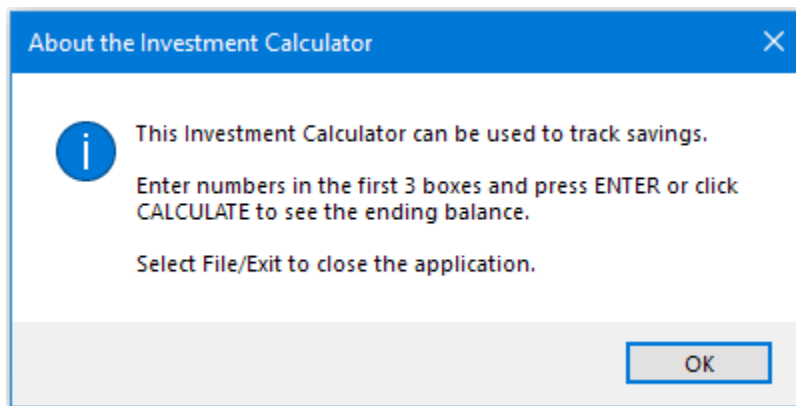
## USING ALERT FOR AN ABOUT DIALOG

In a previous chapter, you created a separate form for an About dialog that could be displayed when you clicked on the Help menu About item. You can also use an ALERT for this task.

Note that the 0x7F character in an ALERT message separates lines within the text.

For example,

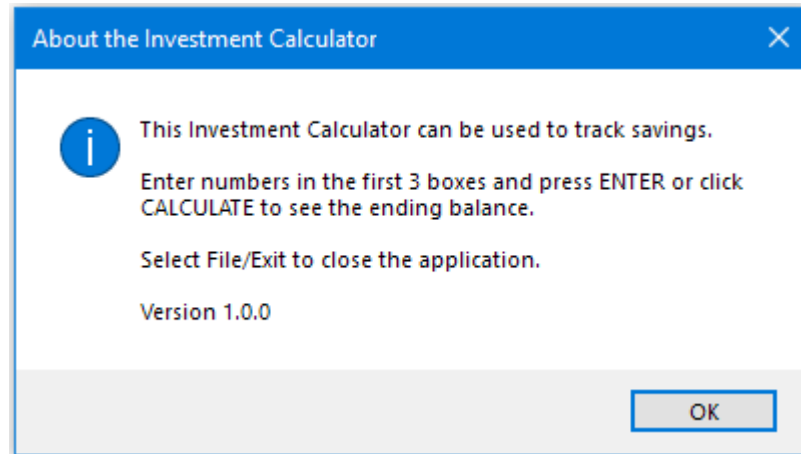
```
Click_mnuHelp
RESULT      INTEGER    1
MESSAGE     INIT       "This Investment Calculator can ":
                        "be used to track savings.":
                        0x7F,0x7F:
                        "Enter numbers in the first 3 boxes ":
                        "and press ENTER or click CALCULATE ":
                        "to see the ending balance.":
                        0x7F,0x7F:
                        "Select File/Exit to close the ":
                        "application."
.
ALERT       NOTE,MESSAGE,RESULT:
           "About the Investment Calculator"
RETURN
```



**Example Alert**

## **EXERCISE - USING ALERT FOR AN ABOUT DIALOG**

1. Open the Investment Calculator and code the Help menu About item to include an appropriate message.
2. Test the program and save it.



## **VALIDATING USER INPUT**

When you allow the user to enter data in an EditText, you cannot always be certain that they will enter valid data. This can obviously pose problems if you are trying to perform arithmetic with this data or if you are trying to save the data in a file that is expecting a certain type of data.

One way to ensure that data is valid for an EditText is to examine the value in the object at execution time. The question is "where is the best place to position the code that will check for valid data?".

EditText objects respond to events such as:

- Change – triggers as each character is typed in the EditText.
- KeyDown – triggers as each character is typed in the EditText but has a code that can be examined to see what key on the keyboard has been pressed.
- KeyPress – same as KeyDown but does not recognize function and navigation keys.
- LostFocus – triggers when the user tabs to or clicks on another control on the form.

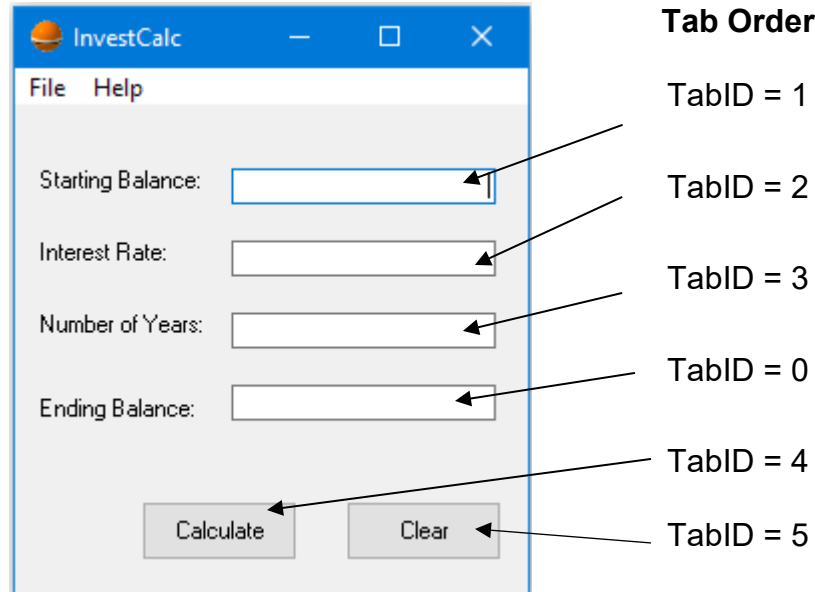
Any of these events could contain validation code for the value in the EditText but each one has certain limitations or drawbacks making it difficult to be foolproof.

## VALIDATING USER INPUT - continued

The `VALIDATE` event for a control works in conjunction with the `CauseValidation` property. It occurs before the focus shifts to a (second) control that has its `CauseValidation` property set to true.

For example, we can modify the Investment Calculator to ensure that the user has entered valid data in each of the first three `EditText` objects.

The first step is to ensure that when the program starts, the focus (the cursor) is in the Starting Balance `EditText` object. When the user presses the `TAB` key on the keyboard, the focus should shift to each of the subsequent `EditText` objects in succession. Use the `TabID` property to ensure the Tab Order of the objects on the form. The `TabID` can be set at design time in the properties window.



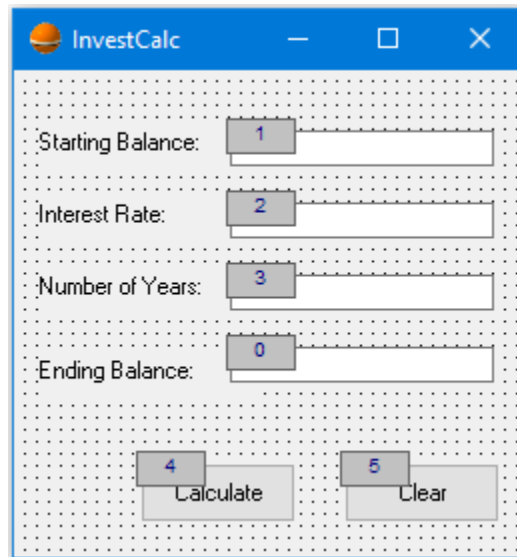
### NOTE

Setting the `TabID` of an object to zero removes it from the tabbing sequence. Since the Ending Balance is calculated, we don't want the user to be able to tab to that field. You can further restrict access to the field by setting the `ReadOnly` property to true.



## VALIDATING USER INPUT - continued

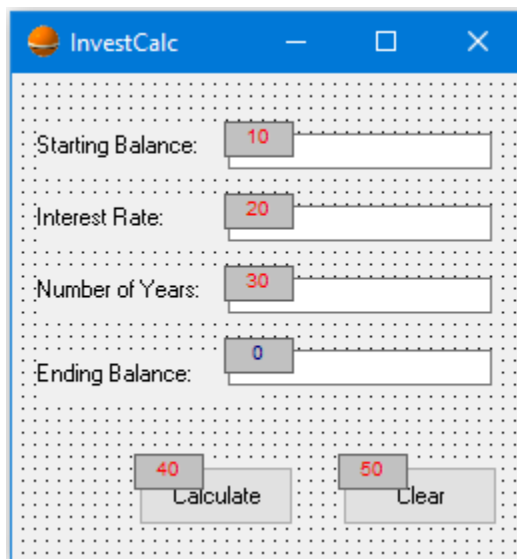
An alternate way of assigning TabIds is a special function of the Designer. Simply select “Start TabID Renumbering from the Tools menu. Once selected, the form will appear as follows:



The screenshot shows a window titled "InvestCalc" with a dotted background. It contains four input fields and two buttons. Each input field has a small gray box to its left containing a blue TabID number. The fields are labeled "Starting Balance:", "Interest Rate:", "Number of Years:", and "Ending Balance:". The buttons are labeled "Calculate" and "Clear".

Field Label	TabID
Starting Balance:	1
Interest Rate:	2
Number of Years:	3
Ending Balance:	0
Calculate Button	4
Clear Button	5

The values shown in blue are the current TabID values. To assign new values, simply click the gray boxes in the order you wish the TabIDs assigned. Double-clicking any box resets all TabIDs and begins numbering with that object.



The screenshot shows the same "InvestCalc" window after renumbering. The TabID numbers in the gray boxes are now red. The "Starting Balance:", "Interest Rate:", and "Number of Years:" fields have TabIDs 10, 20, and 30 respectively. The "Ending Balance:" field has TabID 0. The "Calculate" button has TabID 40 and the "Clear" button has TabID 50.

Field Label	TabID
Starting Balance:	10
Interest Rate:	20
Number of Years:	30
Ending Balance:	0
Calculate Button	40
Clear Button	50

When renumbering is complete, return to the Tools menu and select “Stop TabID Renumbering.

## VALIDATING USER INPUT - continued

The starting TabID number and the TabID increment value are defined on the Options dialog Ordering page.

The screenshot shows the 'Options' dialog box with the 'Ordering' page selected. The left-hand tree view lists various configuration categories, with 'Ordering' highlighted. The main content area is divided into three sections: 'Tab Ids', 'Help Ids', and 'Z Orders'. Each section contains two spinners: 'Starting Value' and 'Increment Value', both of which are currently set to 10. At the bottom of the dialog, there are four buttons: 'OK', 'Cancel', 'Apply', and 'Help'.

**Options Dialog Ordering Page**

## VALIDATING USER INPUT – continued

The next step is to set the initial focus to the Starting Balance EditText object. The SETFOCUS instruction can accomplish this. Place the instruction in the form's INIT event.

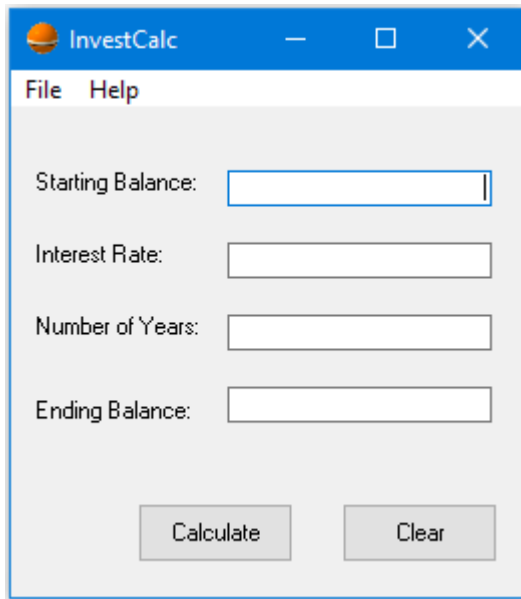
```
Init_frmInvest
    SETFOCUS    txtStartBal
    RETURN
```

Now we need to be certain that the CauseValidation property for each of the objects on the form is set to true. This is the default setting.

To understand the use of the CauseValidation property and the Validate event, consider the following:

**If CauseValidation is set to true for an object before that object can receive focus, the Validate event for the object losing focus is triggered.**

### Example:



CauseValidation is set to true for each EditText object and the Calculate button.

As the user tabs from the Starting Balance EditText to the Interest Rate EditText, the Validate event is triggered for the Starting Balance event.

If the user clicks the Calculate Button while the cursor is in the Number of Years EditText, a Validate event is triggered for the Number of Years EditText.

## VALIDATING USER INPUT – continued

Finally, you need to decide what validation code is necessary for each EditText and code its Validate event appropriately.

For example, we might want to be certain that all of the values in the first three EditText objects are numeric. The TYPE instruction will test a string and set the EQUAL or ZERO flag if it is numeric.

```
Validate_txtStartBal
.
#DATA      DIM      20
#LENGTH    INTEGER  1
.
.Ensure that the data is numeric
.
      GETITEM      txtStartBal,0,#DATA
      TYPE         #DATA
      RETURN       IF EOS                // Empty
      RETURN       IF EQUAL              // Numeric
.
      ALERT        CAUTION,"The starting "::
                  "balance must be numeric.":
                  #Length,"Error"
*
.Non-numeric value entered - highlight the entered text
.
      COUNT        #LENGTH,#DATA
      SETITEM       txtStartBal,1,0
      SETITEM       txtStartBal,2,#LENGTH
      SETFOCUS      txtStartBal          // repeat entry
.
      RETURN
```

## **VALIDATING USER INPUT – continued**

The code on the preceding page:

- Retrieves the value entered into the EditText object.
- Test for a numeric string (TYPE instruction).
- If the string is numeric or null, the user is allowed to move to the new object.
- If the string is not numeric, a message is display using an ALERT box. The data input is then highlighted using the SETITEM instruction and the user is returned to the Starting Balance EditText object by use of the SETFOCUS instruction.

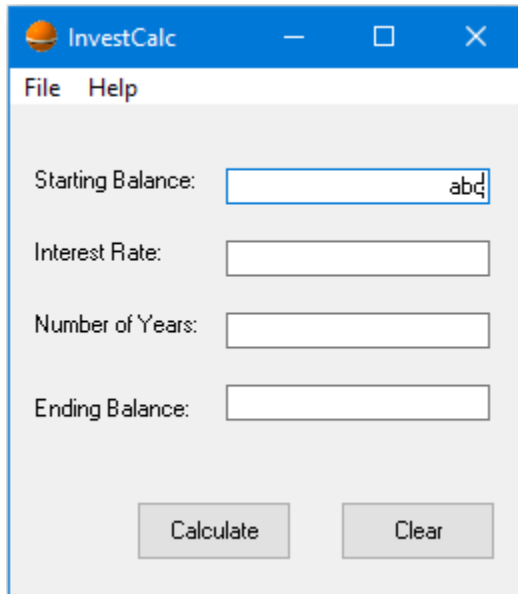
It is good programming practice to highlight the text in an EditText if the text is invalid. SETITEM allows you to specify the starting position for the highlight (in this case 0 or the first byte) and the selected length. Since we want to highlight all of the text entered, we used the COUNT instruction to determine the length of the selection.

With the code in place, the program is started, "abc" is typed in the Starting Balance EditText object, and the user tries to tab to the Interest Rate. The Validate event will trigger for the Starting Balance EditText, a warning message is displayed, "abc" will be highlighted, and control is returned to the Starting Balance EditText object.

By highlighting the invalid code, all the user is required to do is type a new value. As typing begins, the highlighted code will be erased.

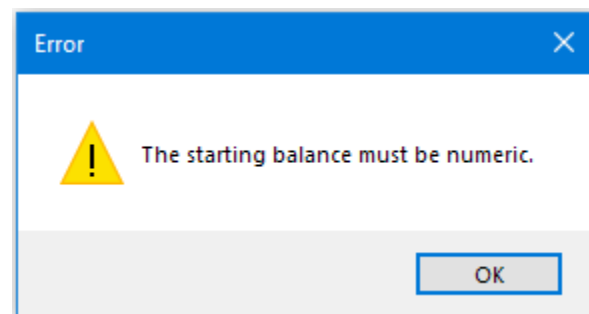
## VALIDATING USER INPUT – continued

Sample Execution:

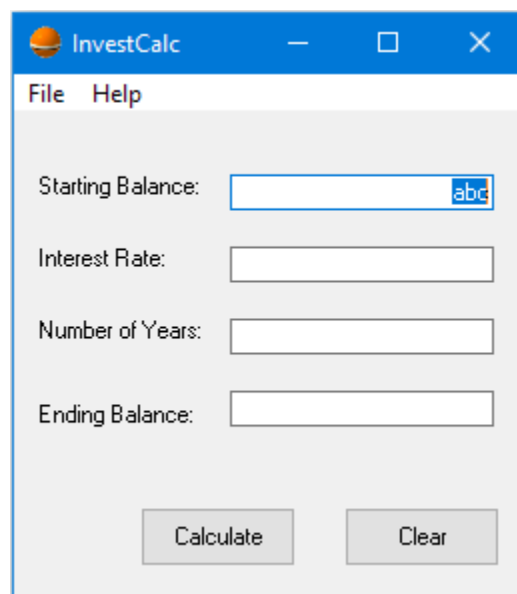


The screenshot shows the 'InvestCalc' application window. It has a menu bar with 'File' and 'Help'. Below the menu bar, there are four input fields: 'Starting Balance:', 'Interest Rate:', 'Number of Years:', and 'Ending Balance:'. The 'Starting Balance' field contains the text 'abc'. At the bottom of the window, there are two buttons: 'Calculate' and 'Clear'.

**Initial entry**



**Within the validation routine**



The screenshot shows the 'InvestCalc' application window after the validation routine has returned. The 'Starting Balance' field now contains the text 'abc' and is highlighted with a blue border, indicating it is the active field. The other fields and buttons remain the same as in the previous screenshot.

**After the validation routine returns**

**VALIDATING USER INPUT – continued**

The code in the Validate event is not executed if the user is never positioned to the EditText object. Therefore, if you have required data (as we do in this Investment Calculator), you must still add code to ensure that all the required fields were entered. For example:

```

MESSAGE    INIT        "You must enter values in the ":
                        " Starting Balance, Interest ":
                        ", and Number of Years fields.":
                        0x7F,0x7F,"Then press Enter or ":
                        "click CALCULATE to see the ":
                        "Ending Balance."

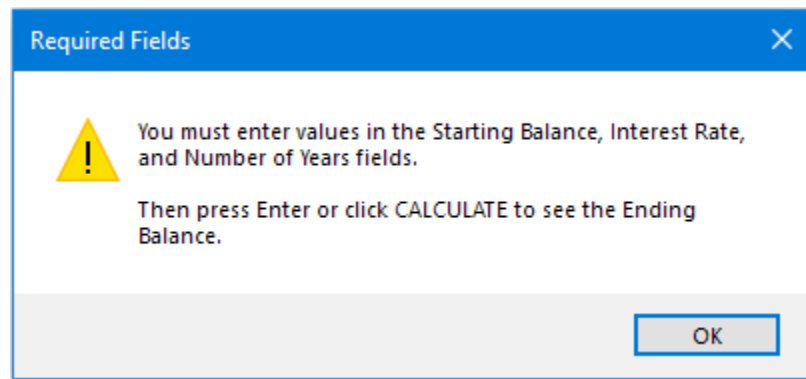
*
.Retrieve the values
.
        CLEAR        StartBal,IRate,Years
.
        GETITEM      txtStartBal,0,DATA
        MOVE          DATA,StartBal
        GETITEM      txtIRate,0,DATA
        MOVE          DATA,IRate
        GETITEM      txtYears,0,DATA
        MOVE          DATA,Years
.
        IF            (StartBal=0 OR IRate=0 OR Years=0)
        ALERT        CAUTION,Message,Response:
                        "Required Fields"
.
        IF            (StartBal = 0)
        SETFOCUS      txtStartBal
        ELSEIF       (IRate = 0)
        SETFOCUS      txtIRate
        ELSE
        SETFOCUS      txtYears
        ENDIF
.
        RETURN
        ENDIF

```

## VALIDATING USER INPUT – continued

This code first zeroes the starting balance, interest rate, and number of years variable. The data from the EditText objects are then retrieved into the appropriate variables. Note that when the DATA retrieved is null, the MOVE instruction does not modify the destination fields. Their contents would be any value left over from a previous iteration of the code. Thus, the CLEAR statement preceding the GETITEMs and MOVEs zeros out these variables.

Next, we test for any missing (i.e., zero) variables. If any are found, we display an ALERT box and set the focus to the first missing field. The user is then required to input a value before calculating the Ending Balance.



**Example ALERT**



## **EXERCISE – INVESTMENT CALCULATOR VALIDATION**

Modify the Investment Calculator as follows:

1. Open the form and set the TabID property for each of the objects. The Ending Balance EditText is for display only and should have its TABID set to 0.
2. Ensure that the CauseValidation property is set to true for each of the objects.
3. Code the Validate event for the first three EditText objects to ensure that the user enters numeric data. Provide the user with a message if the data is invalid and highlight the value in the EditText object.

### **Note**

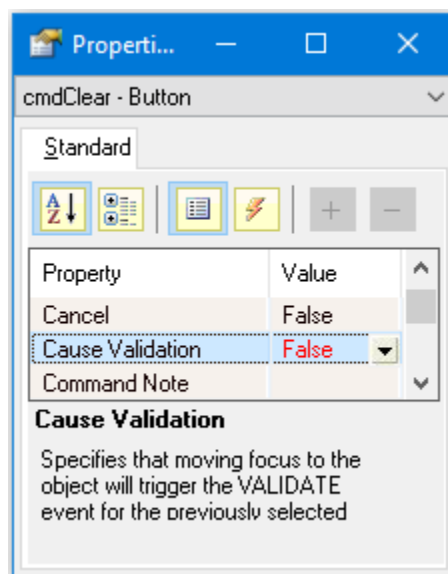
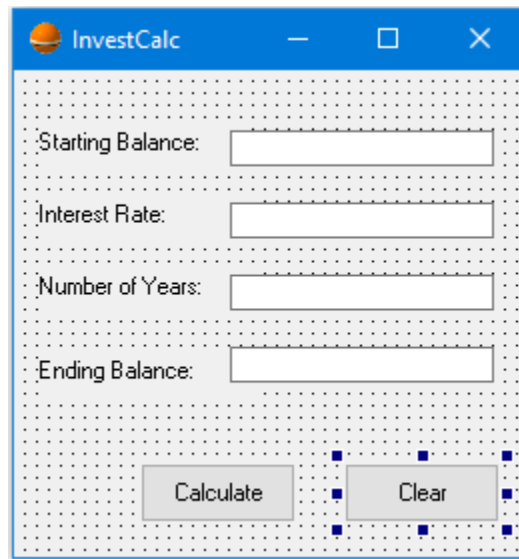
*Be careful to code the correct event for the EditText objects!* When you select an EditText object to open its code window, the default event that Visual PL/B selects is the **Change** event. You must then drop-down the events list and select the **Validate** event.

4. All three input fields are required. Check these EditText objects for data before letting the calculation execute after depressing the Calculate button.
5. Test your program to ensure it is now validating the values placed in each of the EditText objects.

## PREVENTING VALIDATION

There are times when you do not want validation to occur. For example, if you want to add a Quit command button to a form or a Help button, you will want their CauseValidation property to be set to False so that they could be clicked without forcing validation of the data.

If you want to add a button to clear the EditText objects of values, you will not want any values validated because you are going to erase them. To skip validation, simply set the CauseValidation property to False for the Clear button.



## **PREVENTING VALIDATION - continued**

The code for a Clear button on the Investment Calculator form would need to erase the values for each of the EditText objects and set the focus back to the Starting Balance EditText. This can be accomplished with the SETITEM and SETFOCUS instructions.

Click\_cmdClear

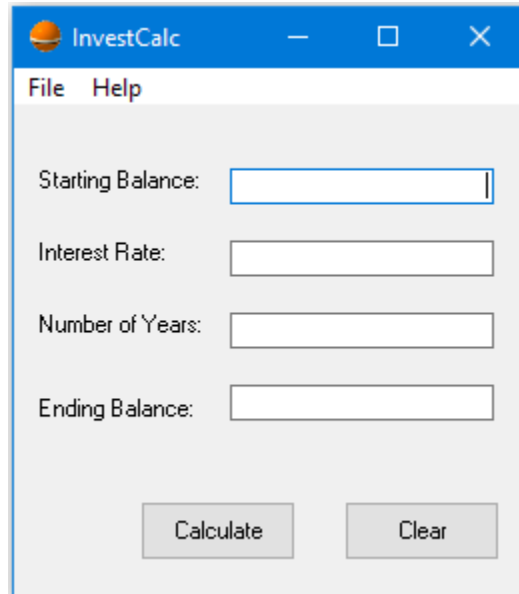
```
.
    SETITEM    txtStartBal,0,""    // clear fields
    SETITEM    txtIRate,0,""
    SETITEM    txtYears,0,""
    SETITEM    txtEndBal,0,""

.
    SETFOCUS   txtStartBal        // Position cursor

.
    RETURN
```

**EXERCISE – ADD A CLEAR BUTTON TO THE CALCULATOR**

1. Open the Investment Calculator form.
2. Add a Button to clear the values from the EditText objects. Set the button's title and name property appropriately.
3. The user should be able to click this button without causing any validation for the EditText objects.
4. Add the appropriate code to the buttons click event to clear the EditText objects and set the focus to txtStartBal.



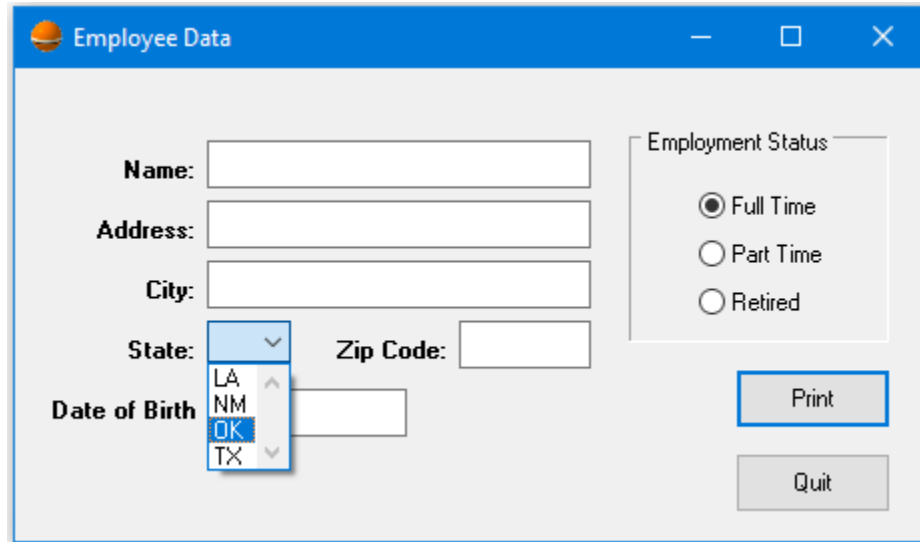
The screenshot shows a Windows-style application window titled "InvestCalc". It has a menu bar with "File" and "Help". The main area contains four labeled text input fields: "Starting Balance:", "Interest Rate:", "Number of Years:", and "Ending Balance:". At the bottom of the form are two buttons: "Calculate" and "Clear".

**Example Form**

## VALIDATING DATA WITH OTHER CONTROLS

There are many properties, controls, and other means to control the values that are entered in a form. On the Investment Calculator, we could eliminate the EditText Validate event logic by simply setting the EditText objects EditType property to decimal. The object itself would then prevent non-numeric data from being entered and thus the Validate event logic would become unnecessary.

In other circumstances, the correct choice of controls can eliminate the need for validation logic. For example, the following form uses a group of radio buttons to allow specification of the Employment Status. When the program executes, the user can only pick from a valid status. This form also requires the selection of a state from a predefined list of valid states.



The screenshot shows a Windows-style application window titled "Employee Data". Inside the window, there is a form with the following elements:

- Name:** A text input field.
- Address:** A text input field.
- City:** A text input field.
- State:** A dropdown menu with a list of states (LA, NM, OK, TX) visible.
- Zip Code:** A text input field.
- Date of Birth:** A text input field.
- Employment Status:** A group box containing three radio buttons: "Full Time" (selected), "Part Time", and "Retired".
- Print:** A button.
- Quit:** A button.

Using objects to circumvent validation code

## **VALIDATING DATA WITH OTHER CONTROLS - continued**

The values in the EditText objects for Name, Address, and City can be any value. You may want to limit the length of these text values. That can be accomplished by setting the Maximum Characters property appropriately.

The Zip Code field needs to be numeric and have the length limited. A combination of the EditType and MaxCharacters properties will satisfy those requirements.

The Date of Birth should be a valid date. You can use the Validate event along some parsing code to ensure that a date is entered.

When validating the entry of Radio buttons in a group, it is normally best to simply set one of the buttons as having a Value property of one and the remainder a Value property of zero. Once the default button is defined, the user will only be able to toggle between valid choices.

Using the Employee Data form for example, you could define variables for each of the fields in the main source file. Next, set the Status variable based on the value of the Radio button group.

```
NAME          DIM          50
ADDRESS       DIM          50
CITY          DIM          50
STATE        DIM           2
ZIPCODE       DIM           5
DOB           DIM           8
STATUS        DIM          10

.
MAIN          PLFORM       employee.PLF

.
              FORMLOAD    MAIN
```

---

## **VALIDATING DATA WITH OTHER CONTROLS - continued**

Next, set the Status variable based in the Click event code of each button.

```
Click_rdoFull
    Move          "Full Time",Status
    RETURN
```

```
Click_rdoPart
    MOVE          "Part Time",Status
    RETURN
```

```
Click_rdoRetired
    MOVE          "Retired",Status
    RETURN
```

If you set the Value property of one of the buttons to one (1), it will be selected by default. In addition, you will not need any validation code in the program.

The Name, Address, City, Zip Code, and Date of Birth EditText objects can be handled in a similar fashion as we handled the EditText objects on the Investment Calculator.

```
Validate_txtBirthDate
.
    GETITEM      txtBirthDate,0,Length
    IF           ZERO
    ALERT        CAUTION,"Enter a birth date.":
                Result,"No Birth Date"
    SETFOCUS     txtBirthDate
    ELSE
    GETITEM      txtBirthDate,0,DOB
    ENDIF
.
    RETURN
```

**VALIDATING DATA WITH OTHER CONTROLS - continued**

Validate\_txtCity

```
.  
    GETITEM    txtCity,0,Length  
    IF         ZERO  
    ALERT      CAUTION,"Enter a city.":  
                Result,"No City"  
    SETFOCUS   txtCity  
    ELSE  
    GETITEM    txtCity,0,City  
    ENDIF  
  
.    RETURN
```

Validate\_txtName

```
.  
    GETITEM    txtName,0,Length  
    IF         ZERO  
    ALERT      CAUTION,"Enter a name.":  
                Result,"No Name"  
    SETFOCUS   txtName  
    ELSE  
    GETITEM    txtName,0,Name  
    ENDIF  
  
.    RETURN
```

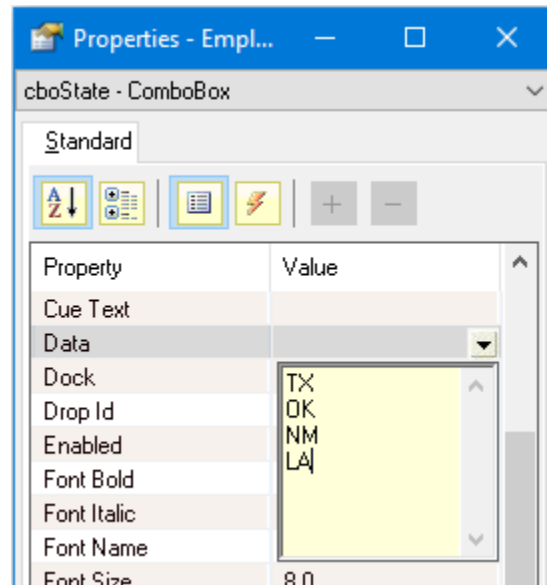
Validate\_txtZipcode

```
.  
    GETITEM    txtZipcode,0,Length  
    IF         ZERO  
    ALERT      CAUTION,"Enter a zip code.":  
                Result,"No Zip Code"  
    SETFOCUS   txtZipcode  
    ENDIF  
  
.    RETURN
```



## **VALIDATING DATA WITH OTHER CONTROLS – continued**

With the ComboBox, you can set the ComboStyle property to "DropDownList" which limits the selection to the items in the list. You must also initialize the list with acceptable values. You can either enter the values in the List property pressing Ctrl+Enter after each Entry or at program execution time using the AddString method.



### **List Data Entry**

If the list is initialized at execution time, you can place the code in the Load event routine of the Form. This event runs after the form is loaded but before it is made visible ensuring that the ComboBox would be populated by the time the user sees the form.

Setting the Sorted property of the ComboBox to True will ensure the items are always in alphabetical order regardless of the order of addition.

```
Load_frmEmployee
```

```
.
    cboState.AddString Using "TX"
    cboState.AddString Using "NM"
    cboState.AddString Using "LA"
    cboState.AddString Using "OK"
.
    RETURN
```

## **VALIDATING DATA WITH OTHER CONTROLS – continued**

When validating the value of this type of ComboBox, the only thing you need to check is whether the user has selected an item from the list.

The GETITEM instruction can retrieve the value of the currently selected item into a numeric variable. If no item is selected, a zero is returned. Otherwise, the value returned may then be used to retrieve the text associated with the selected item.

```
Validate_cboState
```

```
.
    GETITEM    cboState,0,Result          // retrieve index
    IF        ZERO
    ALERT      CAUTION,"Select a State.":
                Result,"No State"
    SETFOCUS   cboState
    ELSE
    GETITEM    cboState,Result,State      // retrieve value
    ENDIF
.
    RETURN
```

If the ComboStyle property is set to CboEdit or CboSimple, the user can enter data in the text portion of the ComboBox instead of choosing from the list. You could then validate the Text property as you did for the EditText objects.

**EXERCISE – CREATE THE EMPLOYEE DATA FORM**

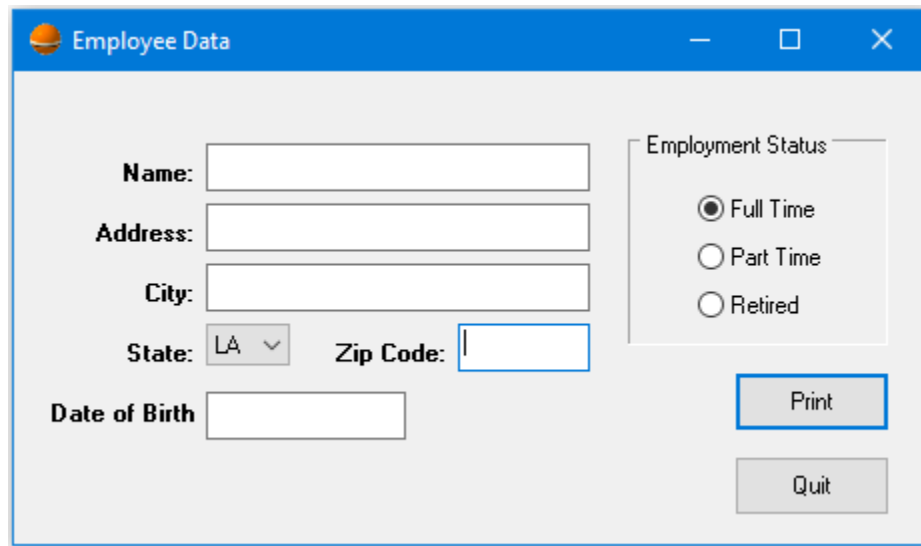
The goal of this application is to create a form that can have information about the employee such as name, address, city, state, zip code, employee status and date of birth.

1. Add a new source file to your current project.
2. Add the following code to the source file:

```
NAME          DIM          50
ADDRESS       DIM          50
CITY          DIM          50
STATE         DIM          2
ZIPCODE       DIM          5
DOB           DIM          8
STATUS        DIM          10
.
MAIN          PLFORM       employee.PLF
.
              FORMLOAD    MAIN
              LOOP
              EVENTWAIT
              REPEAT
```

## **EXERCISE – CREATE THE EMPLOYEE DATA FORM**

3. Use StatTexts, EditTexts, Radios, a GroupBox, a ComboBox, and Buttons to create the following form. Make Print the default button and set the Quit button to terminate the application.



**Example Form**

4. Limit the length of data allowed in the Name, Address, City, and Zip Code fields.
5. Populate the State ComboBox with two-character state abbreviations. You do not have to code all fifty! You can populate the list at design time or execution time.
6. Use the variables defined in the source file to store the information from the EditText objects.
7. Code the appropriate validation routines for each of the controls giving the user error messages when applicable. Collect the values from each of the controls and store them in the source file variables.
8. Each field is required so use code to ensure there is a value for each before printing.
9. Code the Print button to display each variable on the main window.

## **WORKING WITH DATES**

We used some code in the Employee Data form to check and convert date values entered in the Date of Birth EditText object.

There are ways other than an EditText to ensure that the user enters a valid date. One control that accomplishes this is the EditDateTime control. It presents date information where a restricted or specially formatted field is required such as in a payroll or scheduling application. The user may also select a date from a dropdown calendar with a click of the mouse instead of typing a date value.

The EditDateTime control also has properties that can be set to control how the control appears when in use:

- The Text property can be set to a valid date. The default is the current date.
- The MinDate and MaxDate properties restrict the range of dates from which the user can select. The default range is 9/14/1752 to 12/31/9999.
- You can format the date in the display using the Format property. You can also assign a Custom format if you want to display a non-standard date format. For example, d-M-yy displays 18/02/2005.

## **WORKING WITH DATES - continued**

The user can also type a value in the control at run time. If they type a value that is beyond the minimum and maximum dates that were set, the date will default to either the minimum or maximum value depending on whether they were high or low in their entry. The control will not allow invalid characters or dates such as a day of 99.

To retrieve the date chosen, use the Text property.

Since the control will not allow an invalid date, you do not have to code its Validate event. In addition, unless you use the CheckBox style of date display, there will always be a date in the text property. You do not have to check for a missing date. This simplifies your validation. For example, the Validate event for a control called dtpDOB:

```
Validate_edtDOB
    GETITEM    edtDOB, 0, DOB
    RETURN
```

**EXERCISE – MODIFY THE EMPLOYEE DATA FORM**

1. Using the Employee Data form, remove the Date of Birth EditText and replace it with an EditDateTime control. Do not forget to remove the deleted EditText's Validate code.
2. Set the TabID for this control so that is selected after the zip code. You may set any of the custom properties you wish to limit the dates or customize the display.
3. Code the Validate event to retrieve the date entered and store it in your Date of Birth variable.
4. Test the program.

## **CHAPTER NINE**

### ***EVENT PROGRAMMING IN PL/B***





## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on additional Visual PL/B language elements and GUI programming fundamentals.

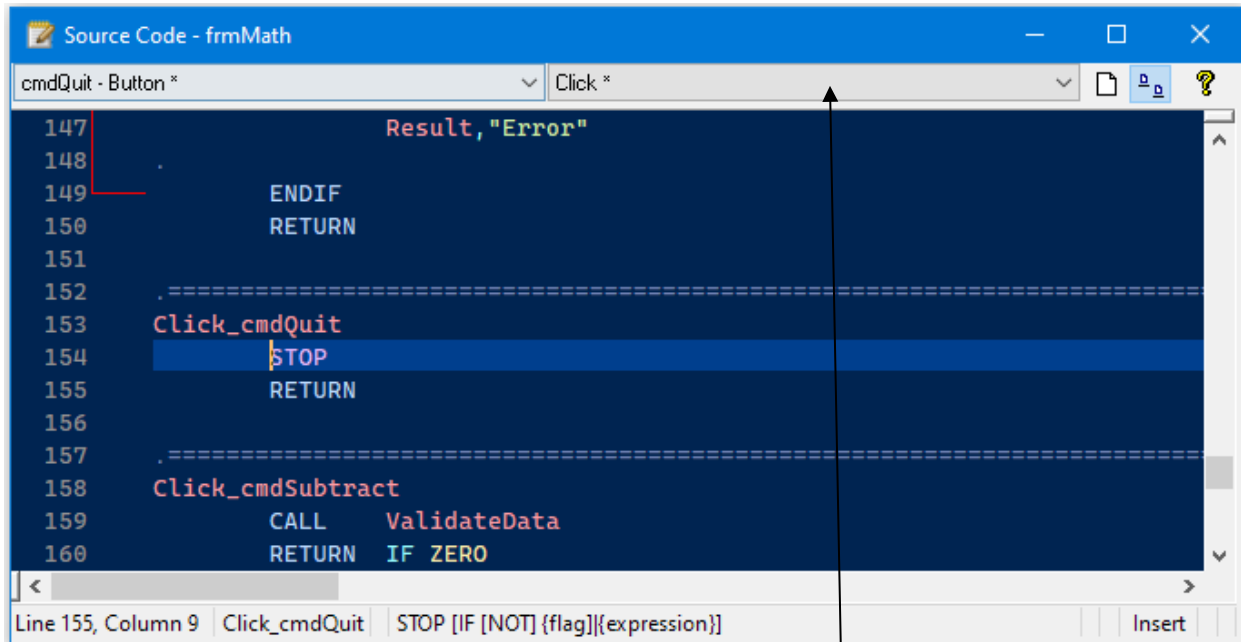
Upon completion of this chapter, you will be able to:

- Understand how events are triggered for objects.
- Demonstrate the use of functions.
- Understand "Pass by Value" function calls.
- Understand "Pass by Reference" function calls.
- Discuss how to declare and process Data Arrays.
- Demonstrate the use of Collection objects.

## UNDERSTANDING EVENTS IN WINDOWS

For each object in Visual PL/B such as EditText and Buttons, there is a predefined set of events that you can respond to by writing code.

When you code the event procedure for a Button, you see all of the events that the Button can recognize by looking at the list of events available in the code window.



**List of Events**

Events in Windows can be classified into several different categories:

- Mouse Events occur when the user presses a mouse button or moves the mouse pointer. Some of the more common mouse events are
  - Click
  - Double-Click
  - MouseDown
  - MouseMove
  - MouseUp

## **UNDERSTANDING EVENTS IN WINDOWS - continued**

- Keyboard Events (KeyPress) occur when a user presses or releases a keyboard key.
- Window Update Events (Paint) occur when a portion of a screen that contains a picture has been overlapped by another window.
- Resizing Events (Resize) occur when a window's size has been changed.
- Activation, Deactivation, Load, Unload, Initialize, and Terminate events are normally triggered when a graphical user interface (GUI) entity has been created, destroyed, or activated by the user switching focus from one object or one window to another. These events include:

Form Load  
Close  
Init Form  
Activate  
Deactivate  
GotFocus  
LostFocus

These events and others are recognized as message in Windows. Window sends the message out to the applications that are running and they pick up the message and process it.

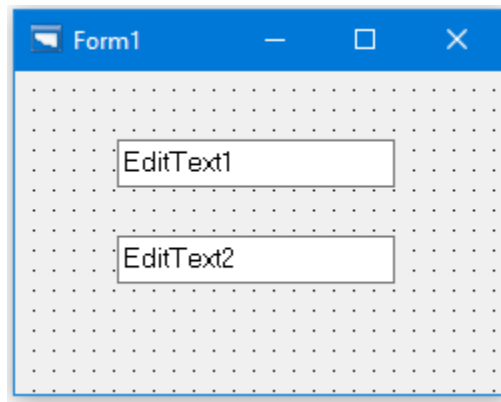
Windows and Visual PL/B handle the recognition of the events. All you have to do is supply the code for what you want to happen when the events occur.

## EXERCISE – EVENT TRIGGERING

1. To illustrate how events are triggered, start a new source file in your current project.
2. Add the following source lines:

```
MAIN          PLFORM      EVENTS.PLF
.
              FORMLOAD   MAIN
              LOOP
              EVENTWAIT
              REPEAT
```

3. Using the Form Designer, create a form with two text boxes. Since this program is for demonstration purposes only, you do not need to change any properties of the form or the textboxes.



4. Open the code window and place DISPLAY statements in each of the following event procedures:

```
Load_Form001
    DISPLAY    "Form Load"
    RETURN

Activate_Form001
    DISPLAY    "Form Activate"
    RETURN
```

**EXERCISE – EVENT TRIGGERING - continued**

```
Change_EditText001
    DISPLAY      "EditText001 Change"
    RETURN

GotFocus_EditText001
    DISPLAY      "EditText001 GotFocus"
    RETURN

KeyPress_EditText001
    DISPLAY      "EditText001 KeyPress"
    RETURN

LostFocus_EditText001
    DISPLAY      "EditText001 LostFocus"
    RETURN

Change_EditText002
    DISPLAY      "EditText002 Change"
    RETURN

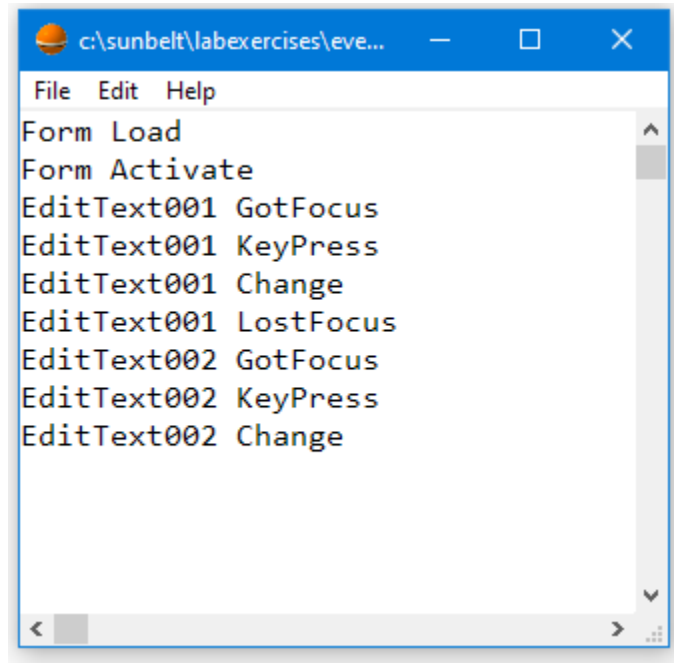
GotFocus_EditText002
    DISPLAY      "EditText002 GotFocus"
    RETURN

KeyPress_EditText002
    DISPLAY      "EditText002 KeyPress"
    RETURN

LostFocus_EditText002
    DISPLAY      "EditText002 LostFocus"
    RETURN
```

## **EXERCISE – EVENT TRIGGERING**

5. Start the program and watch the main program window to see the DISPLAY statements being executed for the various events. Type values in the first and second EditText objects to see the events triggering. This is a great way to see how and when events are fired!



### **Sample Execution**

6. This completes the exercise. You can save the files if you wish but it is not necessary.

## **FUNCTIONS**

A function is a general classification for a block of PL/B statements that are CALLED. The PL/B language refers to these as called event functions, procedures, FUNCTIONS, LFUNCTIONs, and LOADMODs. Each of these is similar in that they are CALLED and they normally RETURN. They differ by allowing parameters and the file in which the code exists. LOADMODs, for example, can be separately compiled programs that contain functions that any program can reference.

Up until now, we have written all of our code as event subroutines. That is to say, they have all been attached to a particular control or the form itself and they were automatically called by Visual PL/B as soon as it recognized the event had occurred. Additionally, the code statements are stored in the form file itself (.PLF).

What about functions that might be common to several controls or events? For these types of functions, Visual PL/B allows us the flexibility of creating our own general functions.

A function tells an application how to perform a specific task. Once the function is defined, it must be called by the application. By contrast, an event function remains idle until called upon to respond to events usually caused by the user such as a click event.

You can place a function in a form module, the main program source file, or an inclusion source file. Generally, it is not a good idea to place functions in forms because they become hidden and are more difficult to locate for the maintenance programmer. You may also encounter problems with undefined variables or objects since the PL/B language does not support forward referencing of data items. Variables and objects must be defined before they are referenced.



## FUNCTIONS – continued

As an example of using a function, you can streamline your Math project that you created and modified in previous chapters. Each of the Buttons has code that retrieves the values from the EditText objects and stores them into variables. We can create a function named "ValidateData" that contains the common code in boldface below:

Click\_cmdSubtract

```
.
    GETITEM    txtValue1,0,Length1
    GETITEM    txtValue2,0,Length2
.
    IF          (Length1 > 0 AND Length2 > 0)
    GETITEM    txtValue1,0,DATA
    MOVE       DATA,Value1
    GETITEM    txtValue2,0,DATA
    MOVE       DATA,Value2
    SUB        Value1,Value2
    MOVE       Value2,DATA
    SETITEM    txtValue3,0,DATA
    SETITEM    lblOperation,0,"-"
    ELSE
    ALERT      CAUTION,"You must enter two values.":
              Result,"Error"
    ENDIF
    RETURN
```

To create a function, return to the IDE and open the program's main source file with the editor. Position to the bottom of the source file and add a label named "ValidateData". Following the label, copy and paste or type the code listed below. To complete the function, add a FUNCTIONEND statement.

The placement of the function code is important since it references objects and variables that must have been previously defined. Thus, positioning the code at the bottom of the main source file ensures that all necessary objects and variable are declared earlier.

## FUNCTIONS – continued

The new function should appear as follows:

```
FORMLOAD  MAIN
LOOP
EVENTWAIT
REPEAT
.
ValidateData FUNCTION
ENTRY
GETITEM   txtValue1,0,Length1
GETITEM   txtValue2,0,Length2
IF        (Length1 > 0 AND Length2 > 0)
GETITEM   txtValue1,0,DATA
MOVE      DATA,Value1
GETITEM   txtValue2,0,DATA
MOVE      DATA,Value2
SETFLAG   NOT ZERO
ELSE
ALERT     CAUTION,"You must enter two values.":
          Result,"Error"
SETFLAG   ZERO
ENDIF
FUNCTIONEND
```

This function retrieves the lengths of the data values from the two EditText objects. If either length is zero, an ALERT dialog is displayed, the zero flag set true, and the function returns. Otherwise, the values of the EditText objects are retrieved into the variables "Value1" and "Value2". The zero flag is then set to False to indicate success and the function returns.

The use of a condition flag to report the status of a called function is more efficient than using a variable.

## FUNCTIONS – continued

With the new function constructed, it is a simple matter to install it in each of the four event functions. Note that the CALL statement invokes (executes) the function and that we check the zero flag to determine whether the operation should proceed.

Click\_cmdSubtract

<b>CALL</b>	<b>ValidateData</b>
<b>RETURN</b>	<b>IF ZERO</b>
.	
SUB	Value2, Value1
MOVE	Value2, DATA
SETITEM	txtValue3, 0, DATA
SETITEM	lblOperation, 0, "-"
.	
RETURN	

**EXERCISE – ADD A FUNCTION TO THE MATH PROJECT**

Your mission is to enhance the Math project by adding a function that can be called to perform data validation.

1. Open the Math project.
2. Add a function to the main source file that will validate the data in the EditText objects like the sample on the previous pages.
3. Replace the code in each of the Buttons that is repeated with a call to the new function.
4. Test the program to ensure it behaves as before and save it.

## PASSING ARGUMENTS TO FUNCTIONS

With ordinary called functions as used in the previous exercise, data is passed to the function and back using globally defined variables. While this approach is simple and efficient, there are times when it would be better to use parameters. For example, let's suppose we have a function that converts a string to all lower case and then capitalizes the first letter. The function is defined as:

```
Convert    FUNCTION
           ENTRY
           LOWERCASE SOURCE, DEST
           XOR      040, DEST
           FUNCTIONEND
```

If we then wanted to convert both the NAME and CITY variables, we would need to call the function using:

```
           MOVE     NAME, SOURCE
           CALL      CONVERT
           MOVE      DEST, NAME
.
           MOVE     CITY, SOURCE
           CALL      CONVERT
           MOVE      DEST, CITY
```

The transfers to and from the variables used by the function are required. Additionally, the SOURCE and DEST variables must be defined before the CALL statements.

## **PASSING ARGUMENTS TO FUNCTIONS - continued**

A better approach is to define the conversion code as a FUNCTION along with a parameter:

```
Convert    FUNCTION
SOURCE     DIM      100
           ENTRY
.
           LOWERCASE SOURCE, DEST
           XOR       040, DEST
           FUNCTIONEND
```

We could then CALL our conversion function for the NAME and CITY variables by using the syntax of:

```
           CALL      CONVERT USING NAME
           MOVE      DEST, NAME
.
           CALL      CONVERT USING CITY
           MOVE      DEST, CITY
```

Now when the CALL is invoked, the variable after the required USING keyword will be moved into the SOURCE variable as part of the transfer of control.

This type of CALL is known as a "Call By Value" because the original variable (NAME or CITY) is not modified by the function itself. The function is working with a copy of the original value.

## PASSING ARGUMENTS TO FUNCTIONS - continued

An even more efficient way of calling functions is a "Call by Reference". With this syntax, there is no moving of the value to the work variable and back. To implement a "Call By Reference", we use pointer variables.

Pointer variables are defined just like ordinary variables but instead of a dimension value, you use a ^. In the case of our example, you might define the SOURCE variable as

```
SOURCE      DIM          ^
```

To implement the use of the pointer variable, we need to change our function as follows:

```
Convert      FUNCTION
SOURCE      DIM          ^
            ENTRY
.
            LOWERCASE SOURCE
            XOR          040, SOURCE
            FUNCTIONEND
```

The CALLs to the function could then be changed to:

```
CALL          CONVERT USING NAME
CALL          CONVERT USING CITY
```

When using the "Call by Reference" (i.e., pointer variables), the function works directly on the NAME variable during the first call and the CITY variable during the second call. Thus, there is no need for a DEST variable or to move data back and forth from work variables.

## COLLECTIONS

Another way that you can work with a group of objects that is like working with arrays is with a Collection object.

The Collection object provides a convenient way to refer to a related group of items as a single object. The items (or members) in a collection are only related by the fact that they exist in the same collection. Unlike arrays, members of a collection do not have to be the same data type.

A Collection is employed by first declaring it:

```
COLL1      COLLECTION
```

Once the collection has been created, members can be added using the LISTINS instruction.

```
GB          GROUPBOX
BUTTON1     BUTTON
BUTTON2     BUTTON
CB1         CHECKBOX
RADIO1      RADIO

.
COLL1      COLLECTION
.
          LISTINS    COLL1,GB,BUTTON1:
                   BUTTON2,CB1,RADIO1
```



## **COLLECTIONS - continued**

Individual members can be removed from the collection though the use of the LISTDEL instruction:

```
LISTDEL    COLL1,BUTTON1,BUTTON2
```

The number of items in a collection is available via the LISTCNT instruction

```
LISTCNT    COUNT,COLL1
```

Individual members of the collection may be retrieved using LISTGET. The member is retrieved into a TYPELESS object since the collection may include any type of object.

Once the collection has been created and members added, the power of a collection becomes apparent. Rather than individually setting a property for each object, you can apply the property setting to all objects in the collection with just a single instruction.

```
SETPROP    COLL1,VISIBLE=1
```

Or

```
SETPROP    COLL1,ENABLED=0
```

## COLLECTIONS – continued

FORMs are also collections and can be processed using the LIST class instructions. The following code retrieves the information for the objects on the Math form when the form is clicked:

```

MouseDown_frmMath
OBJ          VAR          @
VAR1         OBJECT       ^
OBJECTID     FORM         10
OBJTYPE      FORM         7
INFO         DIM          50
COUNT       INTEGER      1
TYPE         DIM          10
INDEX        INTEGER      1
.
      DISPLAY      "Object ID      Type          Info",*N:
                  "-----  -----  ":
                  "-----"
      LISTCNT      COUNT,MAIN
.
      FOR          INDEX FROM 1 TO COUNT

      LISTGET      OBJ,INDEX,MAIN
      TYPE         OBJ,OBJTYPE
.
      SWITCH      OBJTYPE
.
      CASE         "7728"                // Window?
      MOVE         "Window",TYPE
      MOVEADR      OBJ,VAR1
      GETPROP      VAR1,TITLE=INFO
      GETPROP      VAR1,OBJECTID=OBJECTID
.
      CASE         "304"                // Button?
      MOVE         "Button",TYPE
      MOVEADR      OBJ,VAR1
      GETPROP      VAR1,TITLE=INFO
      GETPROP      VAR1,OBJECTID=OBJECTID
.

```

**COLLECTIONS – continued**

```

CASE      "4144"                      // StatText?
MOVE      "StatText",TYPE
MOVEADR   OBJ,VAR1
GETPROP   VAR1,TEXT=INFO
GETPROP   VAR1,OBJECTID=OBJECTID
.
CASE      "1584"                      // EditText?
MOVE      "EditText",TYPE
MOVEADR   OBJ,VAR1
GETPROP   VAR1,TEXT=INFO
GETPROP   VAR1,OBJECTID=OBJECTID
.
DEFAULT
MOVE      OBJTYPE,TYPE
GETPROP   VAR1,OBJECTID=OBJECTID
.
ENDSWITCH
.
DISPLAY   OBJECTID,*H 12,TYPE,*H 24,INFO
.
REPEAT
RETURN
```

## **CHAPTER TEN**

### ***TEXT FILE IO***



## **CHAPTER OVERVIEW AND OBJECTIVES**

The focus of this chapter is to explore reading and writing sequential, text type files.

Upon completion of this chapter, you will be able to:

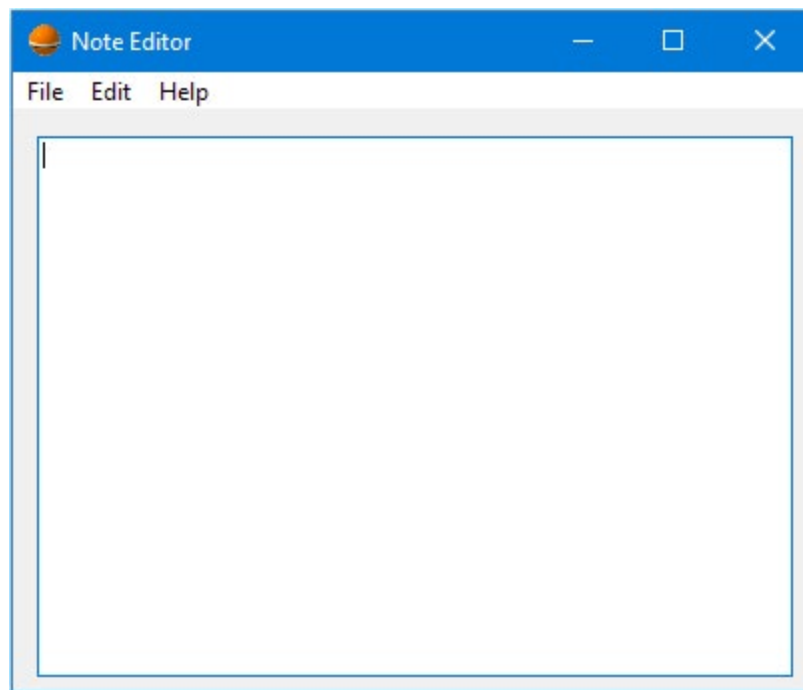
- Demonstrate the use of the OPEN and PREPARE instruction to gain access to open and save dialogs.
- Learn the use of the OPENDEFAULT and PREPDEFAULT instruction to format common dialogs.
- Design and code a Notepad application that opens, reads, and writes text files, uses the Windows clipboard, and the sequential access file statements.

## **CREATING A NOTE EDITOR**

To demonstrate processing of sequential files, we will build a Note Editor. We will also use the Windows clipboard and common file dialogs in this project.

The Note Editor will:

- Edit text messages that are created in an EditText.
- Read text files and place their contents into the EditText object.
- Use the common file dialogs for selection of files to open and save.
- Write text files using the sequential file access methods.



**The Note Editor**

## USING THE WINDOWS CLIPBOARD

Menu items in Visual PL/B can be associated with the standard handling of Edit menu items in Windows applications. This functionality is tied to EditText objects and includes the enabling or disabling of menu items as appropriate, the transfer of data to and from the clipboard, and the deletion of selected data. In addition, the standard Windows shortcut keys are assigned to the items with the exceptions of Select All and Delete.

These special functions include the following:

- **Copy** transfers the selected text to the clipboard leaving the selected text in place in the EditText object.
- **Cut** transfers the selected text to the clipboard removing the selected text from the EditText.
- **Paste** transfers the contents of the clipboard to the insertion point in the EditText object as indicated by the cursor.
- **Delete** removes the selected text from the EditText object.
- **Select All** highlights all text in the EditText object.
- **Undo** cancels the last EditText modification.



## EXERCISE- THE NOTE EDITOR

You will create this project in two stages. First you will create the main portion of the editor by using an EditText object. The user will be able to type messages into the EditText objects, select text in it, copy the text to the clipboard, paste text from the clipboard to the EditText, and clear the contents of the clipboard.

1. Start a new application using the IDE.
2. Add the following code to the main source program.

```
MAIN          PLFORM      EDITOR.PLF
.
      FORMLOAD  MAIN
      LOOP
      EVENTWAIT
      REPEAT
```

3. Create a new form using the Form Designer and set the form's properties to:

<u>Property</u>	<u>Value</u>
Name	frmEditor
Title	Note Editor
WindowType	Primary Fixed

4. Place an EditText on the form arranging it so that it takes up most of the space on the form. Set the object's properties to:

<u>Property</u>	<u>Value</u>
Name	txtNote
Text	(erase all)
Scrollbars	Vertical
MaxLines	32000
WordWrap	True

**EXERCISE- THE NOTE EDITOR - continued**

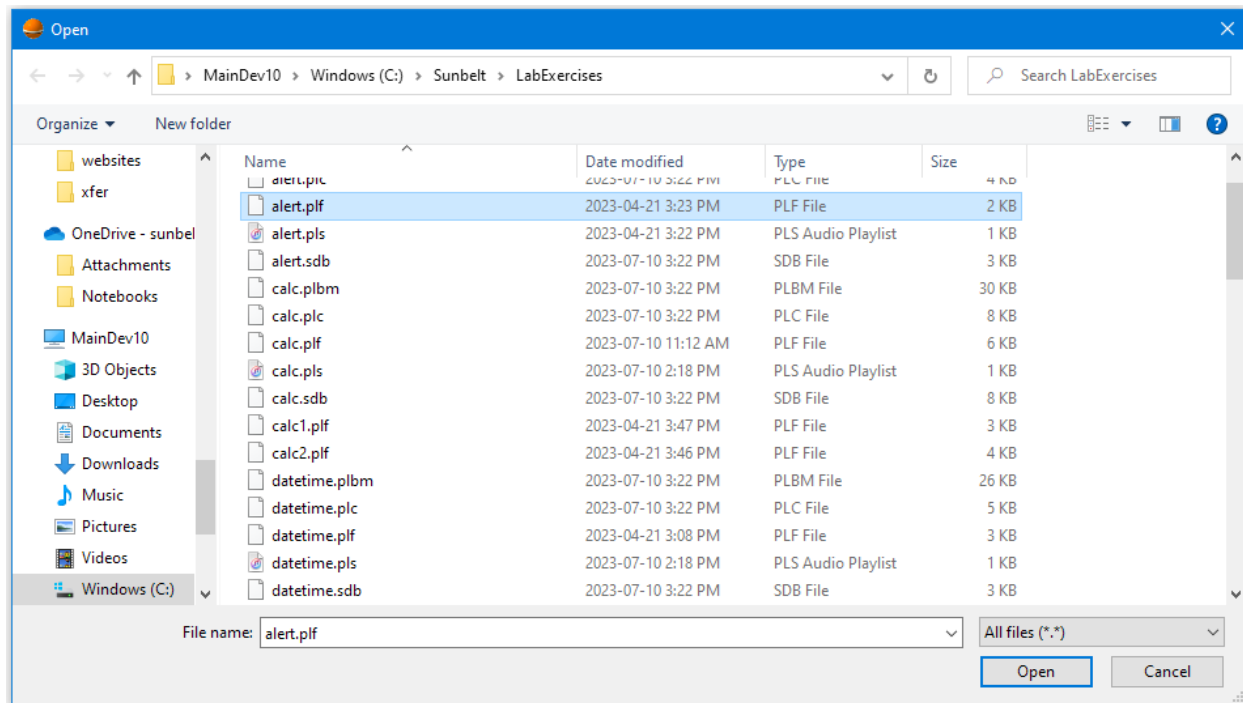
5. Create a menu with the following captions and name:

<b><u>Caption</u></b>	<b><u>Name</u></b>	<b><u>SpecialMenu</u></b>
&File	mnuFile	
E&xit	mnuFileExit	
&Edit	mnuEdit	
Undo	MnuEditUndo	Undo
(separator)	MnuEditSep1	
Cu&t	mnuEditCut	Cut
&Copy	mnuEditCopy	Copy
&Paste	mnuEditPaste	Paste
(separator)	MnuEditSep2	
Select All	MnuEditSelectAll	Select All
Delete	MnuEditDelete	Delete
&Help	mnuHelp	
&About	mnuHelpAbout	

6. Code the File Exit menu item to end the application.
7. Code the Help About menu item to tell the user about the application. You can use an ALERT instruction to do this or you can use a separate form.
8. Test your Note Editor and save it so that we can enhance it in the next exercise. We will be adding functionality to the application that allows read and writing of sequential files.

## COMMON DIALOGS

Visual PL/B has logic included in the OPEN and PREPARE instructions that will invoke the Windows Common Dialog if the file name parameter is a null string. The dialog presented is a familiar one for Windows users and allows navigation to different drives or directories and the selection of a file. The dialog even supports the creation of directories and the deletion of directories and files.



The Windows Common Dialog

## COMMON DIALOGS – continued

The only consideration that must be given when using the common dialog is that a runtime IO error will occur if you user selects "Cancel". This error allows you to use the TRAP mechanism to handle those situations. For instance, the following routine will invoke the common open dialog, process the file, close the file and return. If the user presses "Cancel" while the common dialog is open, we simply return.

FileOpen

```
.
Null      DIM      1
Efile     FILE
.
          TRAP      CancelFile IF IO      // Catch Cancel
          OPEN      Efile,Null            // Show Dialog
          TRAPCLR    IO                    // Clear IO Trap

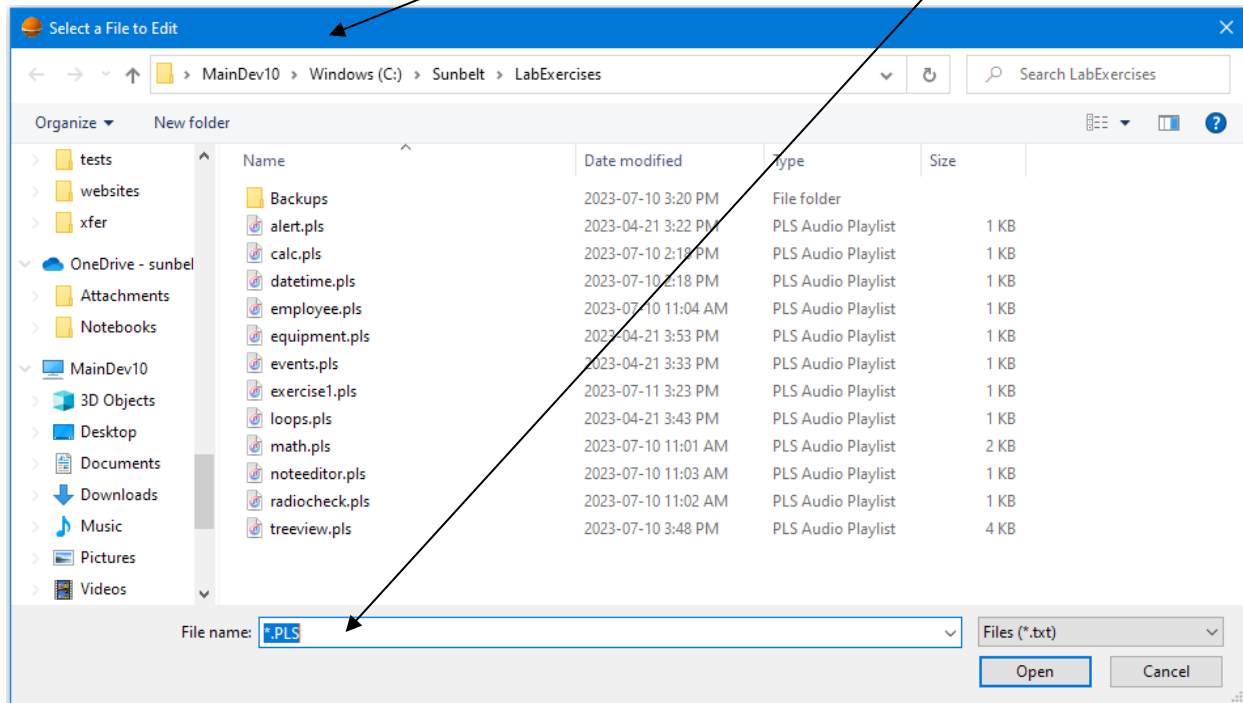
          (processing code)

          CLOSE      Efile
          RETURN                                           // Exit
.
CancelFile
          NORETURN   // Clear the trap entry
          RETURN     // Exit
```

## COMMON DIALOGS – continued

The OPENDEFAULT instruction allows you to customize the common dialog prompt and default file name. The PREPDEFAULT instruction works in a similar manner for the PREPARE common dialog.

OPENDEFAULT "Select a File to Edit", "\*.PLS"



## **EXERCISE – OPENING AND SAVING FILES**

To demonstrate the use of the Common Dialogs, you will modify your Note Editor. Use the Common Dialog to select the file to be opened. Next, you will use sequential file access statements to read and write files.

1. Open the Note Editor project from the previous exercise.
2. Add four new menu items above the Exit item in the File menu. The items to be added are "New", "Open...", "Save...", and a separator. Set the Tag properties of the new items to 1, 2, and 3 respectively.
3. Code the Open menu item to invoke the common dialog. Use sequential file access to read the file and load the data into the EditText object. Sample code to accomplish this is on the following page.
4. Code the Save menu item to write out the edit file. A sample of the code follows the Open sample.
5. Test and save your project.

**EXERCISE – OPEN AND SAVING FILES - continued**

Sample code for the Note Editor:

```
Click_mnuFile
    BRANCH      #EventResult to FileNew:
                FileOpen,FileSave

    STOP

*.....
.Start a new file
.
FileNew
    SETITEM txtNote,0,""
    RETURN

*.....
.Open and Load a file
.
FileOpen
.
EFile      FILE
Null       DIM      1
Seq        FORM      "-1"
EOR        INIT      0x7F
Length     INTEGER   4
Start      INTEGER   4
End        INTEGER   4
Buffer     DIM      ^
*
.Open the File
.
    TRAP      CancelFile IF IO
    OPEN      EFile,Null
    TRAPCLR   IO
*
.Create a Buffer
.
    SMAKE     Buffer,1024
```

**EXERCISE – OPEN AND SAVING FILES – continued**

```
*
.Empty the EditText
.
    SETITEM    txtNote,0,""
*
.Read Records and add them to the EditText
.
    LOOP
    READ      EFile,SEQ;Buffer
    UNTIL     OVER
    SETITEM   txtNote,1,Buffer
    SETITEM   txtNote,1,EOR
    REPEAT
*
.Close the file and exit
.
    SETITEM   txtNote,1,Buffer
    SETITEM   txtNote,1,EOR
    CLOSE     EFile
    RETURN
*
.The Dialog was Cancelled
.
CancelFile
    NORETURN
    RETURN
```



**EXERCISE – OPEN AND SAVING FILES – continued**

```

* .....
.
.File Save
.
FileSave
*
.Create a Buffer large enough to hold the data
.
        GETITEM    txtNote,0,Length
        RETURN     IF ZERO
        SMAKE      Buffer,Length
*
.Retrieve the data into the buffer
.
        GETITEM    txtNote,0,Buffer
*
.Ready the Output File
.
        TRAP       CancelFile IF IO
        PREPARE    EFile,Null
        TRAPCLR    IO
*
.Parse and Write the Records
.
        MOVE       "1",Start
*
.Look for End of Record
.
        LOOP
        SCAN       EOR,Buffer
        WHILE      EQUAL
*
.Isolate the record
.
        BUMP       Buffer,-1
        LENSET     Buffer
        RESET      Buffer,Start

```

**EXERCISE – OPEN AND SAVING FILES – continued**

```
*  
.Output the Record  
.  
        WRITE      EFile,SEQ;*LL,Buffer  
*  
.Position Past the record  
.  
        MOVELPTR   Buffer,Start  
        ADD        "2",Start  
        SETLPTR    Buffer  
        RESET      Buffer,Start  
*  
.Continue Parsing  
.  
        REPEAT     UNTIL EOS  
*  
.Close the File and Exit  
.  
        WEOF       EFile,Seq  
        CLOSE      EFile  
        RETURN
```

## **CHAPTER ELEVEN**

### ***ISAM DATA ACCESS***



## **CHAPTER OVERVIEW AND OBJECTIVES**

This chapter explores performing ISAM data access in VISUAL PL/B

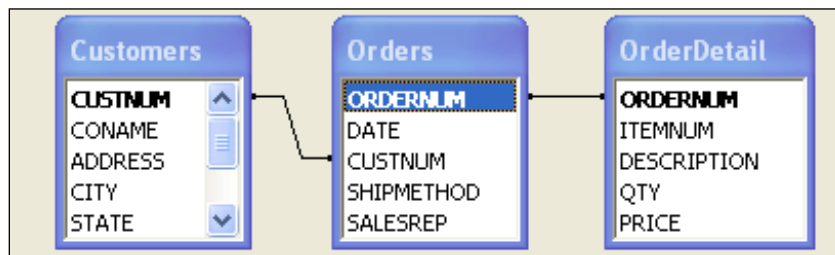
Upon completion of this chapter, you will be able to:

- Construct modular ISAM access routines.
- Attach the ISAM access routines to the appropriate object event.
- Learn how to guide the user using the Enabled property.

## **BUILDING THE SALES APPLICATION**

Many companies use the Indexed Sequential Access Method to access their data stores because ISAM is easy to use and very efficient. ISAM has most of the capabilities found in large and expensive database systems with much less overhead and usually better performance. It remains the logical choice for data access.

In this chapter, you are going to use ISAM for your Sales application files. The mission is to build a Visual PL/B application that allows you to view and modify data in the Sales application's files. This data set contains three files: Customers, Orders, and Detail.



The Customers file contains all the information relating to a customer such as the customer number, name, address, etc. The Customers file is indexed over the customer number field.

Each customer can have any number of orders within the Orders file. This file has basic information about each order such as the date of the customer number, order, salesperson id, and shipping method. The Orders file is indexed over the order number field. The Orders file is related to the Customers file using the customer number.

Each order in the Orders file has one to many records in the OrderDetail table. The OrderDetail table has specific information about each item ordered such as item number, description, quantity, and price. This table is indexed over the order number and the item number. The OrderDetail file is related to the Orders file using the order number.

## **BUILDING THE SALES APPLICATION - continued**

Given that we already have the files defined, our task is to use Visual PL/B to create the user interface for this data.

The scope of this application is to:

1. Allow the required files to be created if they do not exist.
2. Allow the user to view, create, update, and delete records from the Customers file.
3. Allow the user to view, create, update, and delete records from the Orders file for existing customers in the Customer file.
4. Allow the user to view, create, update, and delete records from the Order Detail table for existing orders in the Orders file.
5. Create reports to view all existing customers, orders, and order detail by customer and all orders.

**EXERCISE – BEGIN THE SALES APPLICATION**

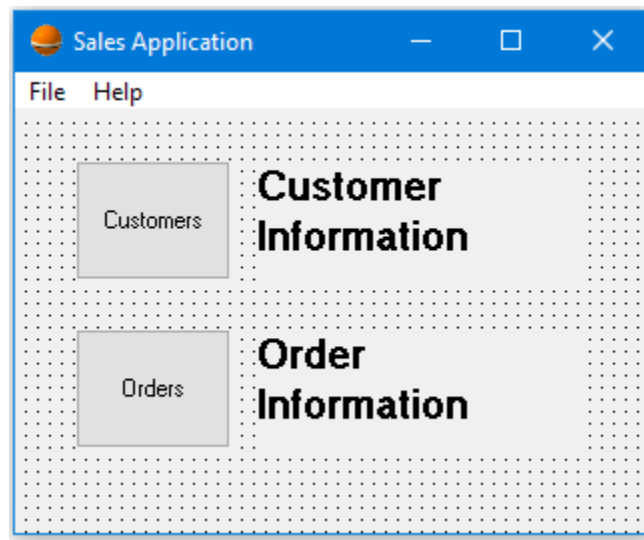
1. Start a new project named "Sales" in Visual PL/B.
2. Add a source file named "Sales" and insert the following code.

```
                INCLUDE    PLBEQU.INC
.
ABOUT          PLFORM    ABOUT.PLF
MAIN            PLFORM    MAIN.PLF
*
.Utility Variables
.
RESULT          INTEGER    4
*
.Execution Begins
.
                FORMLOAD   MAIN
.
                LOOP
                EVENTWAIT
                REPEAT
```



## **EXERCISE – BEGIN THE SALES APPLICATION**

1. Use the Form Designer to create a new form. This form allows navigation through the functionality listed on the previous page.

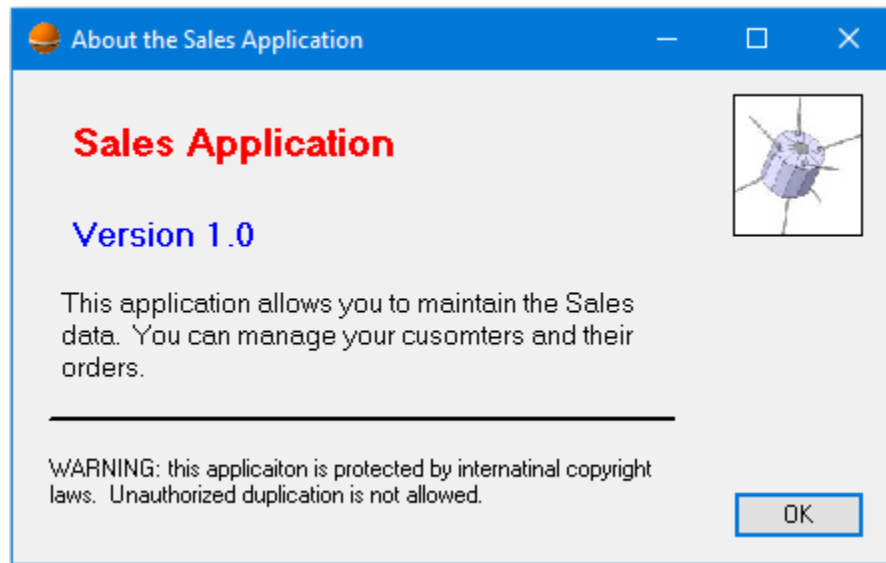


**Sample main form**

2. Place two Buttons and two LabelText objects on the form. Use the graphical style buttons if you wish. We will code the click events in another exercise.
3. Add a File menu that contains an Exit item. Code the Exit item to terminate the application.

## **EXERCISE – BEGIN THE SALES APPLICATION**

4. Add a Help menu that contains an About item. Code the About item to display the program information.



**Sample About Form**

5. When complete, your application should appear as follows:
6. Save your work and test the application as it stands.

## **CREATING THE CUSTOMER FORM**

Accessing ISAM data files with VISUAL PL/B involves the following steps:

1. Create a form to display and navigate through the data.
2. Add the data file and record definitions to the source code.
3. Open or create the data file as the program begins.
4. Access the data file as events are triggered by the user.

We will start by developing the customer maintenance form. This form will

- Display one customer's information at a time.
- Allow new customers to be added.
- Allow modification of an existing customer's data.
- Allow deletion of a customer.
- Provide navigation through the customer file.

## CREATING THE CUSTOMER FORM - continued

Before designing the customer file interface, we should define the fields contained in the customer file. The field definitions for the customer file are below.

Often times, file definitions such as this are placed in individual files and then INCLUDED in each program that requires the definitions. We will use that method in our Sales application.

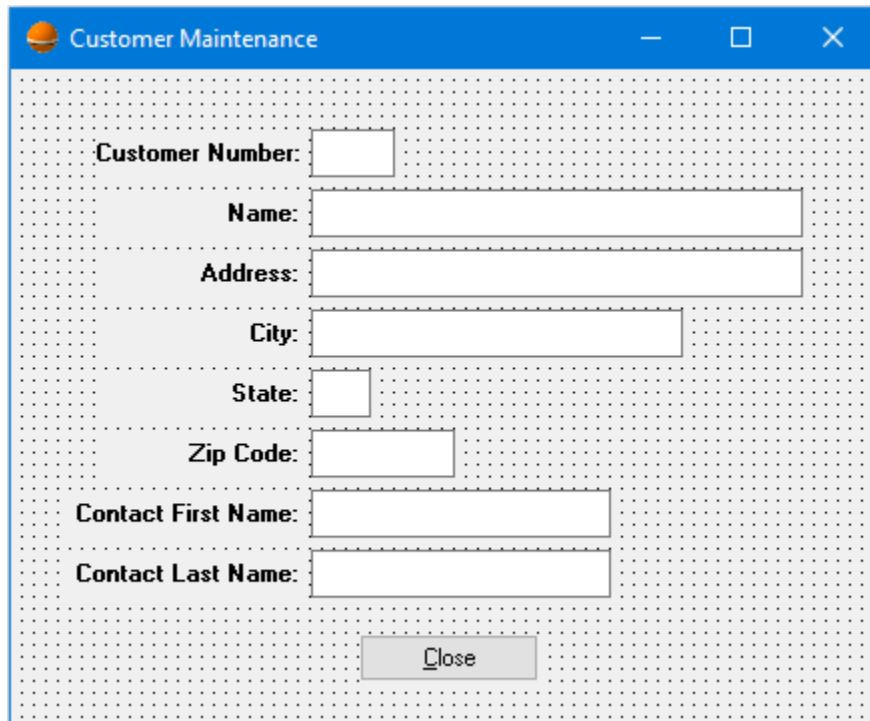
The CUSTFILE definition will be associated with the customer number index file.

```
*
.Customer File Definition
.
CUSTNUM    FORM        10
CONAME     DIM         40
ADDRESS    DIM         40
CITY       DIM         30
STATE      DIM         2
ZIPCODE    DIM         5
CONTACTFN  DIM         20
CONTACTLN  DIM         20
.
CUSTDATA   VARLIST     CUSTNUM, CONAME, ADDRESS, CITY:
                        STATE, ZIPCODE, CONTACTFN, CONTACTLN
.
CUSTKEY    DIM         10
CUSTFILE   IFILE
.
CUSTTEXT   COLLECTION
CUSTCMD    COLLECTION
```

## **CREATING THE CUSTOMER FORM - continued**

Next, we need to design the user interface. Since our form is a maintenance form, each field should be displayed in an EditText object. This will allow the user to input and modify the fields.

Our initial form design might be like this form:



The form is given the title of "Customer Maintenance". Its ObjectPrefix property is set to "cust\_" before any objects are added.

StatText and EditText objects are then drawn on the form for each customer file data field.

Each EditText object is sized in fitting with the defined field size. The EditText objects are given meaningful, unique names and the EditType and MaxChars properties are set appropriately. Finally, the Text properties are cleared of the default data.

A close button is also added to the form.

**EXERCISE – ADD THE CUSTOMER FORM**

1. If you have not done so already, add a new source file to the Sales application. Name the file "customer.inc" and do not add the file as a program when prompted. Place the following definitions in the file.

```
CUSTNUM    FORM        10
CONAME    DIM         40
ADDRESS    DIM         40
CITY       DIM         30
STATE      DIM         2
ZIPCODE    DIM         5
CONTACTFN  DIM         20
CONTACTLN  DIM         20
.
CUSTDATA   VARLIST     CUSTNUM, CONAME, ADDRESS, CITY:
                        STATE, ZIPCODE, CONTACTFN, CONTACTLN
CUSTKEY    DIM         10
CUSTFILE   IFILE
```

**EXERCISE – ADD THE CUSTOMER FORM**

2. Next, we need to add some local variables and the customer form and file definitions in our source program (sales.pls). Modify the source code by adding the boldfaced lines.

```

                INCLUDE    PLBEQU.INC
                INCLUDE    CUSTOMER.INC
*
.Utility Variables
.
RESULT      INTEGER      4
YESNO        INTEGER      1, "0x24"
ADDING      INTEGER      1
MSG         DIM          25
NWORK10     FORM         10
DIM10       DIM          10
.
CUSTOMER    PLFORM       CUSTOMER.PLF
ABOUT       PLFORM       ABOUT.PLF
MAIN         PLFORM       MAIN.PLF
.
                FORMLOAD   CUSTOMER
                FORMLOAD   MAIN
.
                LOOP
                EVENTWAIT
                REPEAT

```

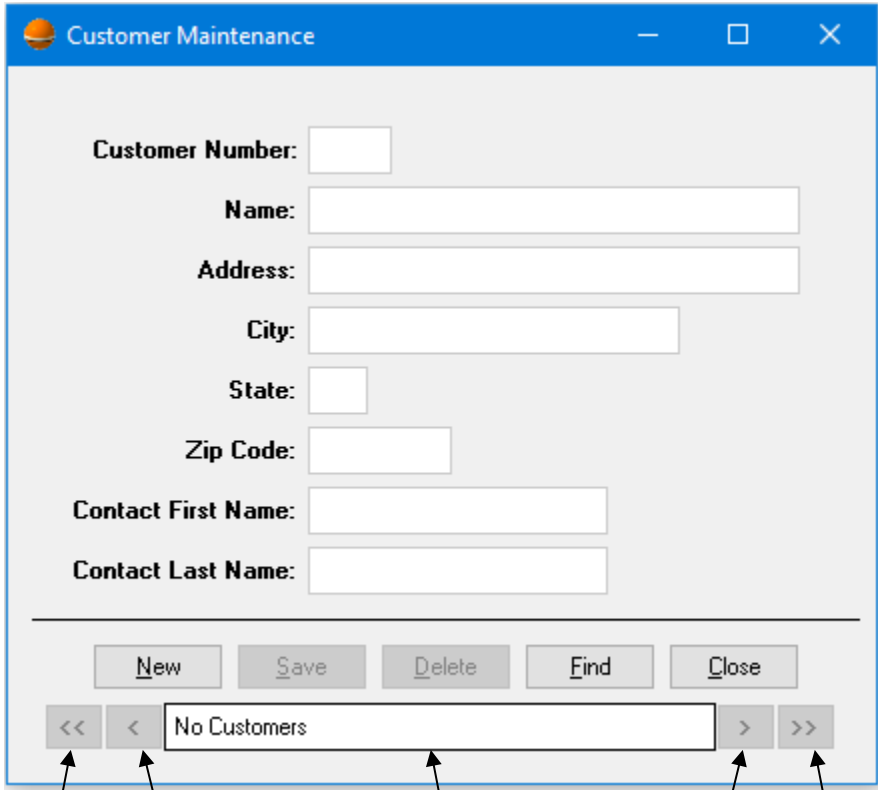
**EXERCISE – ADD THE CUSTOMER FORM**

3. Define a customer form similar to the one on the preceding page. Set the form's Visible property to False.
4. Set the EditText objects' Name, EditType, MaxChars, and Text properties as required for each field.
5. Set the Customer Number EditText object's ReadOnly property to true.
6. Specify the Text properties for the accompanying StatText labels.
7. Assign the TabId properties of each object to a meaningful sequence.
8. Code the Close button to set the customer form's Visible property to False.
9. Add code to the Main form's customer button to set the customer form's Visible property to True.
10. Save the forms and the source files.
11. Compile and test the application.



## ADDING ISAM ACCESS

Once the form has been designed and added to the application, we must next add our ISAM access code. Our goal is a form that appears as follows:



The screenshot shows a window titled "Customer Maintenance" with a blue header bar. The main area contains a form with the following fields:

- Customer Number:
- Name:
- Address:
- City:
- State:
- Zip Code:
- Contact First Name:
- Contact Last Name:

Below the form is a horizontal bar containing several buttons and a text field:

- Buttons: New, Save, Delete, Find, Close
- Navigation buttons: <<, <, >, >>
- Text field: No Customers

Arrows point from the following labels to the corresponding elements in the navigation bar:

- First Record: points to the << button
- Previous Record: points to the < button
- Number of Customers: points to the "No Customers" text field
- Next Record: points to the > button
- Last Record: points to the >> button

## MODULAR DESIGN

This code for this form is constructed in a modular form for ease of understanding and simplified maintenance. We will take advantage of other PL/B language features such as VARLISTs and Sunbelt's Simplified File IO to keep the code clean and efficient.

The goal is to encapsulate each function required (open, read, delete, etc) in a separate routine. We will then call the routines as needed based upon operator requests.

This code is placed in a separate source file and INCLUDED in our Sales application just as the customer file data definitions were.

We do make a conscious effort to keep the code in the source file and not in the form. Placing code in a form has two disadvantages:

1. It requires more steps to locate a particular source line when it is buried in a form.
2. You will end up with some of the code in the form and some in the source file and always have to guess where the line you need is.

By keeping all the code in the source file, there is never a question where to look. All we need in the form is a call to the correct routine such as:

```
Click_cust_cmdDelete
      CALL      CustDelete
      RETURN
```

## **ISAM ACCESS ROUTINES**

Note that much of the code deals with controlling the objects available to the user at any given point in the program. A principal of GUI design is to guide the user through the process. Rather than having a Delete button that when selected displays a message saying "No customer selected", it would be better to just not enable the Delete button if there was no customer. This "steering" of the customer saves many times more lines of code than it costs.

The routines needed are:

<b><u>Routine</u></b>	<b><u>Purpose</u></b>
<b>CustOpen</b>	Open the Customer file and creates the necessary Collection objects.
<b>CustPrep</b>	Creates the Customer file.
<b>CustClose</b>	Close the Customer file.
<b>CustFirst</b>	Position to the first customer record.
<b>CustLast</b>	Position to the last customer record.
<b>CustPrevious</b>	Position to the previous customer record.
<b>CustNext</b>	Position to the next customer record.
<b>CustNew</b>	Add a new blank customer record
<b>CustSave</b>	Save the current customer record.
<b>CustDelete</b>	Delete the current customer record
<b>CustGet</b>	Retrieve object values into record fields
<b>CustPut</b>	Store record fields into objects
<b>CustCount</b>	Retrieve the Customer file record count and display.
<b>CustVerify</b>	Enable Save button if all required fields are present.

## OPEN ROUTINE

We begin by defining the ISAM routines with the Open routine. This routine named "CustOpen" will associate the disk file name with the IFILE variable. It is called by the form's Load event. We'll include logic to optionally create the data and ISAM file if it doesn't exist.

```
* .....
.
.Build a collection of the EditText objects
. and a collection of the command buttons
. except New and Close.
.
CustOpen
    LISTINS    CUSTTEXT,cust_txtNumber:
                cust_txtName, cust_txtAddress:
                cust_txtCity,cust_txtState:
                cust_txtZipcode,cust_txtFName:
                cust_txtLName
.
    LISTINS    CUSTCMD,cust_cmdSave,cust_cmdDelete:
                cust_cmdFirst,cust_cmdPrevious:
                cust_cmdNext,cust_cmdLast
*
.Open the Customer File
.
    TRAP      CustPrep IF IO
    OPEN      CUSTFILE,"CUSTOMERS"
    TRAPCLR   IO
*
.Attempt to read the first record
.
    CALL      CustFirst
    CALL      CustCount
    RETURN
```

## PREPARE AND CLOSE ROUTINES

Next, we will add the file creation routine. This routine will ask the user whether the file should be created. A response of "No" will terminate the application. Otherwise, the file is created. This routine is called via the TRAP in the CustOpen routine.

```
* .....
.
.Create the Customer File
.
CustPrep
    ALERT      TYPE=YESNO,"The Customer file "::
               "does not exist - Create It ?":
               RESULT,"Warning"
    STOP      IF (RESULT = $IDNO)
.
    PREPARE    CUSTFILE,"CUSTOMERS","CUSTOMERS":
               "1-10","167"
    RETURN
```

The CustClose routine is called when the customer form closes. The customer file is never actually closed since it is opened by the form's Load event and the form is never destroyed. An implicit close occurs when the program terminates. All that is necessary is to perform the Delete routine if case the program was in the middle of adding a new customer. The form itself is then hidden.

```
* .....
.
.Close the customer form
.
CustClose
    CALL      CustDelete IF (Adding)
    SETPROP   frmCustomer,VISIBLE=$FALSE
    RETURN
```

## READ ROUTINE

Next, we will add the record retrieval routines. These routines read the ISAM file and store the values into the correct fields. The "CustRead" routine retrieves and displays the customer identified by CUSTNUM. If CUSTNUM is zero, the first customer is retrieved.

```
* .....
.
.Read the Customer File by Customer Number
.
CustRead
    CALL      CustFirst IF (CUSTNUM = 0)
.
    MOVE      CUSTNUM,CUSTKEY
    TRAP      CustRead1 IF IO
    READ      CUSTFILE,CUSTKEY;CUSTDATA
    TRAPCLR   IO
    RETURN    IF OVER
.
    CALL      CustPut
.
CustRead1
    RETURN
```

## FIRST AND LAST RECORD ROUTINES

We next add a routine to read the first customer record. If a record is retrieved, the EditText objects are filled and enabled to allow modification. The command and navigation buttons are also enabled.

```
* .....
.
.Read the First Customer Record
.
CustFirst
    FILL      " ",CUSTKEY
    READ      CUSTFILE,CUSTKEY;;
    READKS    CUSTFILE;CUSTDATA
.
    IF        OVER
    SETPROP   CUSTTEXT,ENABLED=$FALSE
    SETPROP   CUSTCMD,ENABLED=$FALSE
    RETURN
    ENDIF
.
    CALL      CustPut
    RETURN
```

Now comes the CustLast routine that retrieves the last customer (by number) from the file.

```
* .....
.
.Read the Last Customer Record
.
CustLast
    FILL      "9",CUSTKEY
    READ      CUSTFILE,CUSTKEY;;
    READKP    CUSTFILE;CUSTDATA
    RETURN    IF OVER
.
    CALL      CustPut
    RETURN
```

## PREVIOUS AND NEXT RECORD ROUTINES

We also need routines to move to the next and previous record from our current position in the file:

```

* .....
.
.Read the Previous Customer Record
.
CustPrevious
    READKP    CUSTFILE;CUSTDATA
    IF        OVER
    ALERT     NOTE,"Beginning of file.":
              RESULT,"Move Previous"
    CALL      CustFirst
    RETURN
    ENDIF
.
    CALL      CustPut
    RETURN
* .....
.
.Read the Next Customer Record
.
CustNext
    READKS    CUSTFILE;CUSTDATA
    IF        OVER
    ALERT     NOTE,"End of file.":
              RESULT,"Move Next"
    CALL      CustLast
    RETURN
    ENDIF
.
    CALL      CustPut
    RETURN

```



## NEW RECORD ROUTINE

When the New button is click, we enter an "adding" mode. In this routine, we do the following:

1. Set a switch to indicate addition mode
2. Enable the EditText objects and the Delete button
3. Disable the New and Save buttons
4. Set the focus on the first field (number)
5. Erase any values in the EditText objects

```
* .....
.
.Add a Customer Record
.
CustNew
    SET          ADDING
    SETPROP     CUSTTEXT,ENABLED=$TRUE
    SETPROP     cust_txtNumber,READONLY=$FALSE
    SETPROP     cust_cmdDelete,ENABLED=$TRUE
    SETPROP     cust_cmdNew,ENABLED=$FALSE
    SETPROP     cust_cmdSave,ENABLED=$FALSE
    SETPROP     CUSTCMD,ENABLED=$FALSE
    SETFOCUS    cust_txtNumber
    DELETEITEM  CUSTTEXT,0
    RETURN
```

## SAVE ROUTINE

The Save routine is employed when adding new records and updating existing ones. The "adding" switch indicates the program's current state. Sunbelt's Simplified IO allows the record to be written using a key extracted from the data.

If we are in addition mode, we must remember to clear the mode, update the count of customers, and enable the New button again.

```
* .....
.
.Save a Customer Record
.
CustSave
        CALL      CUSTGET
.
        IF        (ADDING)
        WRITE     CUSTFILE;CUSTDATA
        CALL      CustCount
        CLEAR     ADDING
        SETPROP   cust_cmdNew,ENABLED=$TRUE
        SETPROP   cust_txtNumber,READONLY=$TRUE
        SETPROP   CUSTCMD,ENABLED=$TRUE
        ELSE
        UPDATE    CUSTFILE;CUSTDATA
        ENDIF
.
        RETURN
```

## DELETION ROUTINE

The CustDelete routine begins by detecting the current program's state.

- If we were in the process of adding a record, the new record is abandoned, the previously displayed record is read and displayed, and the New button is enabled.
- If we were not in the process of adding a new record, the currently displayed record is deleted. To maintain a valid file position, we then move to the next record. Should that move fail, we move to the previous record. The count of customers is then updated.

```
* .....
.
.Delete the Customer Record
.
CustDelete
    IF          (Adding)
    CLEAR       CUSTDATA, Adding
    SETPROP     cust_cmdNew, ENABLED=$TRUE
    SETPROP     CUSTCMD, ENABLED=$TRUE
    SETPROP     cust_txtNumber, READONLY=$TRUE
    CALL        CustPut
    CALL        CustRead
    ELSE
    RETURN      IF (CUSTNUM = 0)
    DELETE      CUSTFILE
    CALL        CustNext
    CALL        CustPrevious IF OVER
    CALL        CustCount
    ENDIF
.
    RETURN
```

## COUNT ROUTINE

A routine is next added to display the current number of customers in the StatText object at the bottom of the screen. Care is taken to construct a pleasing message. If there is more than one customer, the navigation keys are also enabled.

```
* .....
.
.Update the count of Customers
.
CustCount
        GETFILE      CUSTFILE,RECORDCOUNT=NWORK10
.
        IF           (NWORK10 = 0)
        MOVE         "No Customers",MSG
        ELSEIF       (NWORK10 = 1)
        MOVE         "1 Customer",MSG
        ELSE
        MOVE         NWORK10,DIM10
        SQUEEZE      DIM10,DIM10
        PACK         MSG WITH DIM10," Customers"
        ENDIF
.
        SETITEM      cust_lblCount,0,MSG
.
        IF           (NWORK10 > 1)
        SETPROP      CUSTCMD,ENABLED=$TRUE
        ENDIF
        RETURN
```

## PUT ROUTINE

Our next step in adding ISAM access to the application is to create two routines that will transfer the record values to and from the form objects.

CustPut will transfer data from the record variables into the objects.

```
* .....
.
.Transfer Record Data to the Form Objects
.
CustPut
    IF          (CUSTNUM > 0)
    MOVE        CUSTNUM,CUSTKEY
    ELSE
    CLEAR       CUSTKEY
    ENDIF
.
    SETITEM     cust_txtNumber,0,CUSTKEY
    SETITEM     cust_txtName,0,CONAME
    SETITEM     cust_txtAddress,0,ADDRESS
    SETITEM     cust_txtCity,0,CITY
    SETITEM     cust_txtState,0,STATE
    SETITEM     cust_txtZipcode,0,ZIPCODE
    SETITEM     cust_txtFName,0,CONTACTFN
    SETITEM     cust_txtLName,0,CONTACTLN
    RETURN
```

## GET ROUTINE

The CustGet routine will move the data from the EditText objects to the record variables.

```
* .....
.
.Transfer Record Data from the Form Objects
.
CustGet
    GETITEM    cust_txtNumber,0,Result
    IF         ZERO
    CLEAR      CUSTNUM
    ELSE
    GETITEM    cust_txtNumber,0,CUSTKEY
    MOVE       CUSTKEY,CUSTNUM
    ENDIF
.
    GETITEM    cust_txtName,0,CONAME
    GETITEM    cust_txtAddress,0,ADDRESS
    GETITEM    cust_txtCity,0,CITY
    GETITEM    cust_txtState,0,STATE
    GETITEM    cust_txtZipcode,0,ZIPCODE
    GETITEM    cust_txtFName,0,CONTACTFN
    GETITEM    cust_txtLName,0,CONTACTLN
.
    RETURN
```

**DATA VERIFY ROUTINE - continued**

One last routine is needed to complete the implementation of ISAM access with the form. This routine ensures that any required fields are present before enabling the Save button.

```
* .....
.
.Enable the Save button when the required
.fields are input
.
CustVerify
*
.Assume failure
.
        SETPROP    cust_cmdSave,ENABLED=$FALSE
*
.Verify the Customer Number
.
        GETITEM    Cust_txtNumber,0,RESULT
        RETURN     IF ZERO
*
.Verify the Customer Name
.
        GETITEM    Cust_txtName,0,RESULT
        RETURN     IF ZERO
*
.All required fields present
.
        SETPROP    cust_cmdSave,ENABLED=$TRUE
        RETURN
```

## CALLING ACCESS ROUTINES FROM FORMS

With the definition of the ISAM access routines complete, we are ready to implement them. As all of the actual code to manage the file access is in the routines we have constructed, we only need to CALL them from the correct event procedure in the form.

We begin by coding the form's load and close events:

```
Load_frmCustomer
    CALL      CustOpen
    RETURN
```

```
Close_frmCustomer
    CALL      CustClose
    RETURN
```

Next, we will code the command buttons:

```
Click_cust_cmdNew
    CALL      CustNew
    RETURN
```

```
Click_cust_cmdSave
    CALL      CustSave
    RETURN
```

```
Click_cust_cmdDelete
    CALL      CustDelete
    RETURN
```

```
Click_cust_cmdClose
    CALL      CustClose
    RETURN
```



## **CALLING ACCESS ROUTINES FROM FORMS - continued**

Now we can code the navigation buttons.

```
Click_cust_cmdFirst  
    CALL      CustFirst  
    RETURN
```

```
Click_cust_cmdPrevious  
    CALL      CustPrevious  
    RETURN
```

```
Click_cust_cmdNext  
    CALL      CustNext  
    RETURN
```

```
Click_cust_cmdLast  
    CALL      CustLast  
    RETURN
```

Finally, we will attach our required field logic to the Validate events of the customer name and number fields.

```
Validate_cust_txtName  
    CALL      CustVerify  
    RETURN
```

```
Validate_cust_txtNumber  
    CALL      CustVerify  
    RETURN
```

**EXERCISE – ADDING ISAM ACCESS TO THE FORM**

In this exercise, we will add controls and code to the Sales application Customer maintenance form.

1. Add a new source file named "customer.pls" to the Sales application. Do not add it as a source program.
2. Place the code on the preceding pages into the file.
3. Add a line to the bottom of the Sales application main source file (sales.pls) as follows:

```
LOOP  
EVENTWAIT  
REPEAT
```

.

```
INCLUDE    CUSTOMER.PLS
```

4. Add the navigation and control buttons as required.

## **CHAPTER TWELVE**

### ***ADVANCED DATA ACCESS TECHNIQUES***

## **CHAPTER OVERVIEW AND OBJECTIVES**

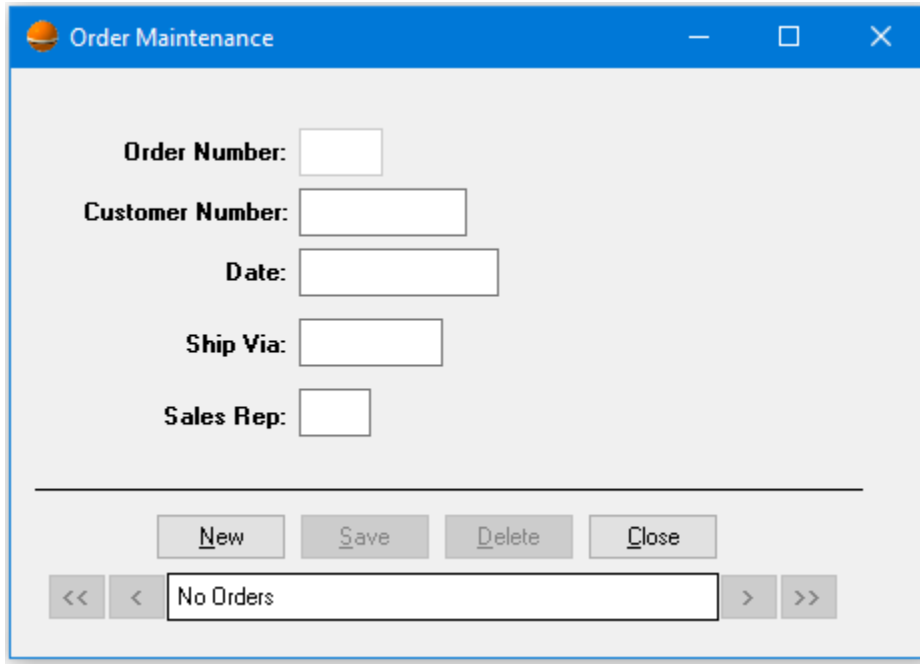
This chapter focuses on the use of standard objects that allow viewing and modifying related data in a Visual PL/B application.

Upon completion of this chapter, you will be able to:

- Use a ComboBox to allow selection of a valid customer and sales rep for an order.
- Define a set of Radio Buttons for the order shipment method.
- Use an EditDateTime object to allow selection of a valid order date.
- Use a ListView object to view and select order detail records.
- Add, modify, and delete order items using a right mouse click.
- Use EditNumber objects to retrieve order detail data.

## THE ORDERS MAINTENANCE FORM

You completed the customer maintenance form in the previous chapter for the Customer file. Now you need to create a form that allows the user to view, modify, add, and delete order information for each customer. To do this we will copy the customer form's components and change the variable names and object properties.

The image shows a screenshot of a software window titled "Order Maintenance". The window has a blue title bar with standard Windows window controls (minimize, maximize, close). The main area is light gray and contains five labels with corresponding text input fields: "Order Number:", "Customer Number:", "Date:", "Ship Via:", and "Sales Rep:". Below these fields is a horizontal line. Under the line are four buttons: "New", "Save", "Delete", and "Close". At the bottom of the window is a status bar with navigation buttons "<<", "<", ">", and ">>". In the center of the status bar is a text box containing the text "No Orders".

**Example Order Form**

## EXERCISE – CREATE THE ORDERS FORM

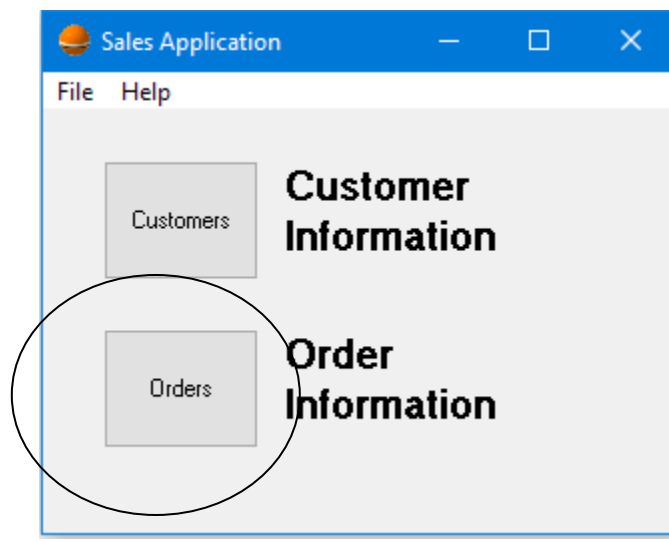
To create the object form quickly, we will simply use the customer form as a template. This is easier than starting fresh but the choice is yours.

1. Begin by creating the data definition inclusion file named "order.inc". Use the "customer.inc" file as a template. When complete, the file should appear as:

```
*
.Order File
.
ORDNUM      FORM      10
ORDCUST     FORM      10
ORDDATE     DIM        8
ORDSHIP     FORM       1
ORDSALES    FORM       3
.
ORDDATA     VARLIST    ORDNUM,ORDCUST,ORDDATE:
                  ORDSHIP,ORDSALES
.
ORDKEY      DIM        10
ORDFILE     IFILE
.
ORDTEXT     COLLECTION
ORDCMD      COLLECTION
```

## **EXERCISE – CREATE THE ORDERS FORM - continued**

2. Copy the customer form named customer.plf to order.plf. This can be done easily by opening the customer form and using "Save As" to save it with a name of "order.plf".
3. Set the order form's properties as appropriate for the order data. You will need to change the form name (frmOrder) and title (Orders Maintenance) and the object prefix value (ord\_).
4. Update the text properties of the StatText objects to reflect the order field names.
5. Set each EditText object's name to begin with "ord" instead of "cust". Also set the EditType and MaxChars properties as needed.
6. Ensure that the TabId properties are set to a meaningful order.
7. Change the prefix of all remaining objects from "cust" to "ord".
8. Change the called routines in the code window to refer to the order file routines.
9. Modify the main form to set the frmOrder object's visible property to true when the orders button is clicked.



**EXERCISE – CREATE THE ORDERS FORM - continued**

10. Add the following boldfaced lines to the beginning of the sales.pls source file:

```

        INCLUDE      PLBEQU.INC
        INCLUDE      CUSTOMER.INC
        INCLUDE      ORDER. INC
*
.Utility Variables
.
RESULT      INTEGER      4
YESNO       INTEGER      1, "0x24"
ADDING      INTEGER      1
MSG         DIM          25
NWORK10     FORM         10
DIM10       DIM          10
.
CUSTOMER    PLFORM       CUSTOMER.PLF
ORDER      PLFORM       ORDER. PLF
ABOUT      PLFORM       ABOUT. PLF
MAIN        PLFORM       MAIN. PLF
.
        FORMLOAD      CUSTOMER
        FORMLOAD      ORDER
        FORMLOAD      MAIN
.
        LOOP
        EVENTWAIT
        REPEAT
.
        INCLUDE      CUSTOMER.PLS
        INCLUDE      ORDER. PLS

```

11. Compile and test the program.

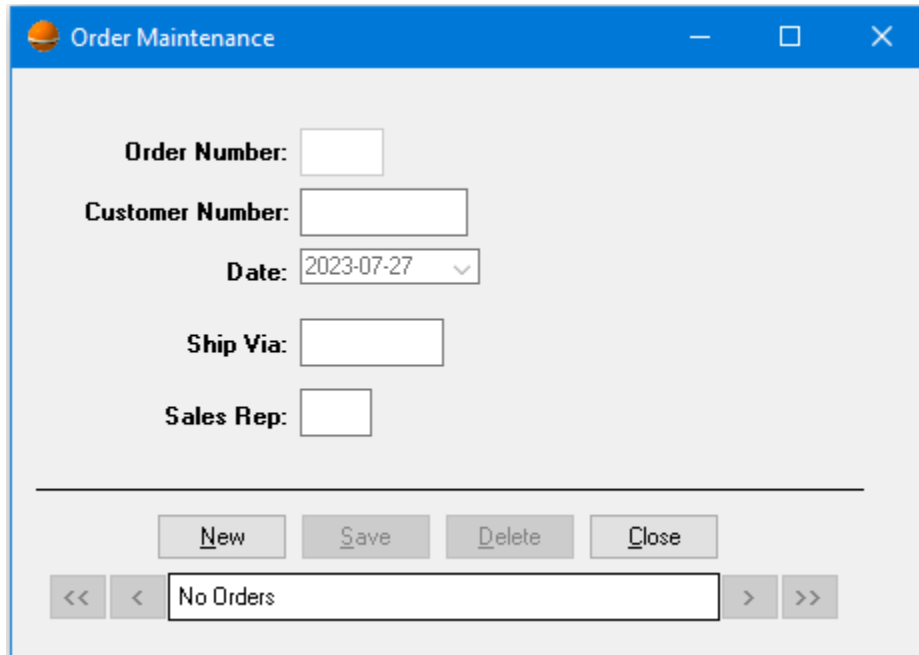


## ENHANCING THE ORDER FORM

While operational, our order form has some serious shortcomings.

1. You must know the customer number to enter an order.
2. The input of the order date could be made easier.
3. The selections of the shipment method and salesman could be improved.

An important part of GUI programming is correct object selection. By selecting the correct object, both the job of the user and that of the programmer can be simplified. For example, the current order form requires the input of a date but there is no indication to the user regarding the format. Likewise, the programmer will be required to accommodate any data the user enters. Rather than using an EditText in this case, a better choice would be the EditDateTime object.

The image shows a Java Swing window titled "Order Maintenance". Inside the window is a form with five labeled text input fields: "Order Number:", "Customer Number:", "Date:", "Ship Via:", and "Sales Rep:". The "Date:" field is currently displaying "2023-07-27" and has a small downward arrow on its right side, indicating it is an EditDateTime object. Below these fields is a horizontal line, and underneath that line are four buttons: "New", "Save", "Delete", and "Close". At the very bottom of the window is a status bar containing navigation arrows (<<, <, >, >>) and a text area that says "No Orders".

**Form with EditDateTime object**

## ENHANCING THE ORDER FORM - continued

A default date is provided by the EditDateTime control. It also is aware of the date's format should the user wish to type in a date. Finally, the user can select a date by clicking the arrow and accessing the calendar.

The screenshot shows a window titled "Order Maintenance" with a blue header bar. Inside the window, there are several input fields and buttons. The "Date:" field is set to "2023-07-27" and has a dropdown arrow. A calendar pop-up is displayed over the "Date:" field, showing the month of July 2023. The calendar has a grid with days of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat) and dates (1-31). The date "27" is highlighted with a blue border. Below the calendar, there is a button labeled "New" and a status bar showing "<< < No Orders > >>".

Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Today: 2023-07-27

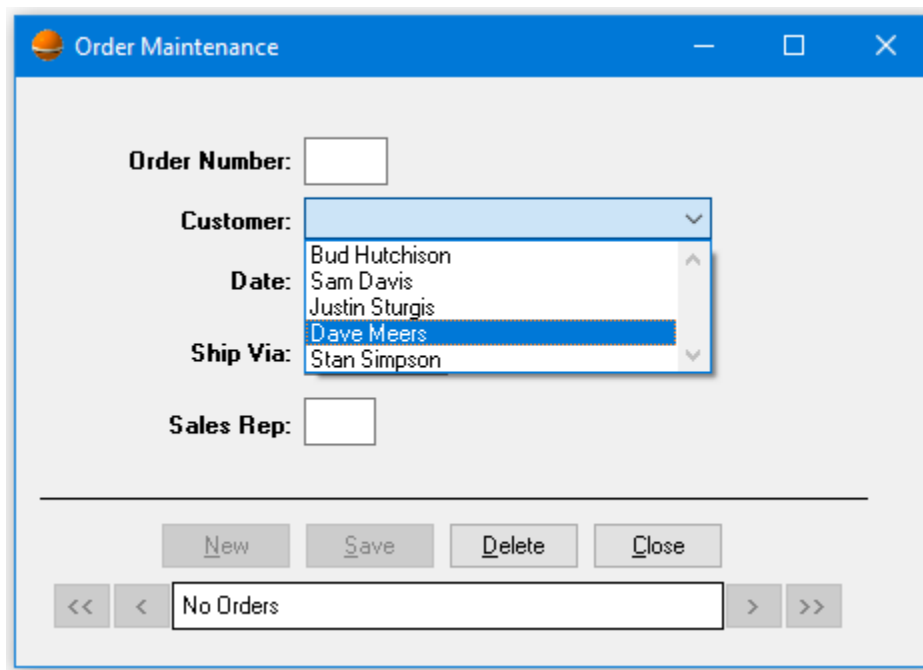
**Example of Date Selection**

The EditDateTime object has several properties that ensure flexibility with most needs. These properties include the date's format and a number of appearance values.

## **ENHANCING THE ORDER FORM - continued**

In a similar fashion, it is not reasonable to require the user to know each customer's assigned number during order maintenance. Rather than using an EditText for the customer number, we will use a ComboBox.

We will fill the ComboBox with customer names in alphabetical order each time the application starts. When a customer is selected, we will retrieve the customer number. The customer number will be hidden from the user. We will even change the label to just read "Customer".

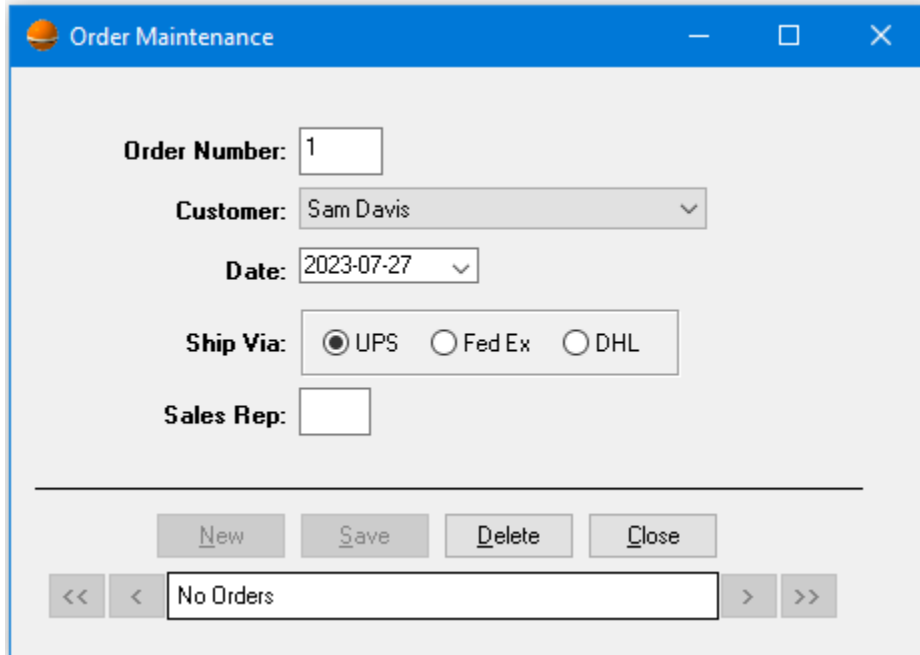


The screenshot shows a Windows-style application window titled "Order Maintenance". Inside the window, there are several labels and input fields: "Order Number:" followed by a text box, "Customer:" followed by a ComboBox, "Date:" followed by a text box, "Ship Via:" followed by a text box, and "Sales Rep:" followed by a text box. The ComboBox is open, displaying a list of customer names: "Bud Hutchison", "Sam Davis", "Justin Sturgis", "Dave Meers" (which is highlighted in blue), and "Stan Simpson". Below these fields, there is a horizontal line. Under the line, there are four buttons: "New", "Save", "Delete", and "Close". At the bottom of the window, there is a status bar with navigation buttons "<<", "<", ">", and ">>". In the center of the status bar is a text box containing the text "No Orders".

**Example of Customer Selection via a Combobox**

## ENHANCING THE ORDER FORM - continued

Next, we will modify the Ship Via input field. This field is defined as a numeric value. Let's use radio buttons to allow the user a selection of shipment methods. We will set the default to UPS. The use of radio buttons and a default value means we will need no verification code for this field.



The screenshot shows a window titled "Order Maintenance" with a blue header bar. The window contains the following fields and controls:

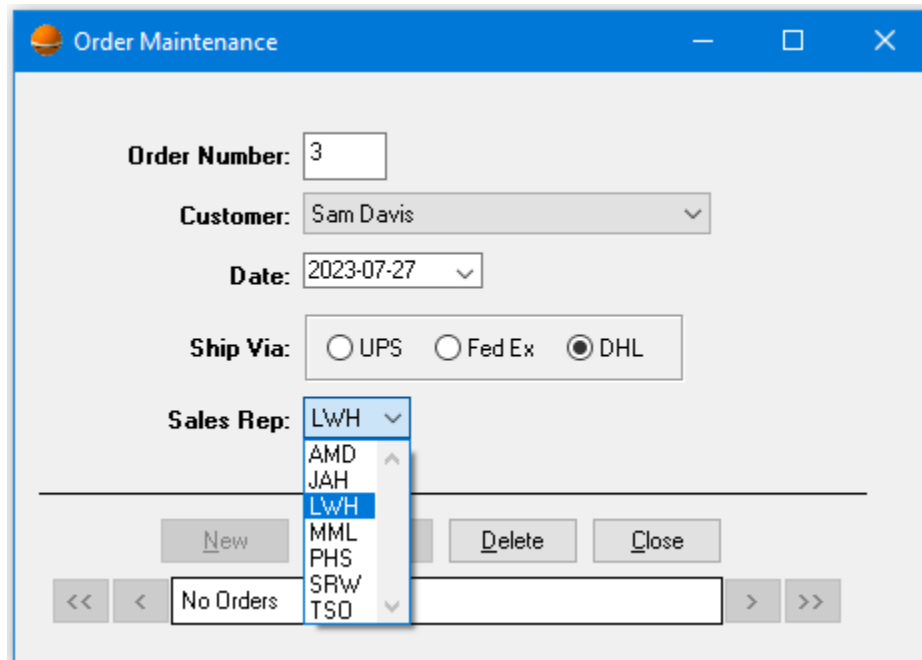
- Order Number:** A text input field containing the value "1".
- Customer:** A dropdown menu showing "Sam Davis" with a downward arrow.
- Date:** A date picker showing "2023-07-27" with a downward arrow.
- Ship Via:** A group box containing three radio buttons: "UPS" (selected), "Fed Ex", and "DHL".
- Sales Rep:** A text input field.

Below the fields, there is a horizontal line and a row of four buttons: "New", "Save", "Delete", and "Close". At the bottom, there is a navigation bar with four buttons: "<<", "<", ">", and ">>". In the center of the navigation bar is a text box containing "No Orders".

**Example of Ship Via selection using Radio Buttons**

## **ENHANCING THE ORDER FORM - continued**

Finally, for the Sales Rep field, let's use another ComboBox. We can fill this list at design time with all the sales reps initials.

The screenshot shows a Windows application window titled "Order Maintenance". It contains several form fields: "Order Number" with a text box containing "3", "Customer" with a dropdown menu showing "Sam Davis", "Date" with a date picker showing "2023-07-27", and "Ship Via" with three radio buttons: "UPS", "Fed Ex", and "DHL" (which is selected). Below these is the "Sales Rep" field with a dropdown menu open, showing a list of initials: "LWH", "AMD", "JAH", "LWH", "MML", "PHS", "SRW", and "TSD". The "LWH" option is currently selected. At the bottom of the window, there are buttons for "New", "Delete", and "Close", along with a status bar showing "<< < No Orders > >>".

**Example of Sales Rep selection via a ComboBox**

That will complete the redesign of the order form. The careful use of graphical controls makes data entry more intuitive for the user while removing much of the burden of data validation for the programmer.

## **EXERCISE – ENHANCING THE ORDERS FORM**

In this exercise, we will replace customer number and Sales Rep EditText with ComboBoxes. We will also change the date's EditText to an EditDateTime object and the Ship Via's EditText to a set of Radio Buttons.

1. Delete the existing customer number, date, ship via, and sales rep EditText objects from the orders form.
2. Add a ComboBox named "ord\_cboCustnum" for the customer field. The ComboBox's combostyle property should be set to DropDown List and the Sorted property should be True.
3. Insert an EditDateTime object for the date input. Set the object name to "ord\_edtDate". Set the format property to Short Date.
4. Add a GroupBox with three Radio Buttons for the Ship Via. Label the buttons "UPS", "FedEx", and "DHL" and set their names to "ord\_rdoUPS", "ord\_rdoFedEx", and "ord\_rdoDHL" respectively. Assign all three Radios a GroupId property of one (1) and set the Value property of the UPS radio to one.
5. Add a DropDown List style ComboBox for the sales rep field. Add several sets of three letter initials to the List property. Name the object "ord\_cboSalesRep" and set the Sorted property to True.
6. Arrange the TabId properties of the new objects into a meaningful sequence.
7. These modifications require no changes to the field definition file "order.inc".

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

8. Open the "order.pls" inclusion file and modify the OrdOpen routine. The ORDTEXT collection items have new names. The code should read:

```
LISTINS    ORDTEXT,ord_txtNumber:
           ord_edtDate,ord_rdoUPS:
           ord_rdoFedEx,ord_rdoDHL:
           ord_cboSalesRep
```

9. Next, we need to create a new routine named "OrdCust". The purpose of this routine is to fill the customer ComboBox with existing customers. This routine is called during the OrdOpen routine and we will also call it to from Customer form to ensure that the list is always accurate. Note that we use the SetItemData method to associate the customer number with the shown customer name. Before returning, we ensure that the ISAM file is positioned to the record of the customer shown.

```
* .....
.Load the customer ComboBox
.
OrdCust
    DELETEITEM ord_cboCustnum,0
.
    FILL      " ",CUSTKEY
    READ      CUSTFILE,CUSTKEY;;
.
    LOOP
    READKS    CUSTFILE;NWORK10,CONAME
    UNTIL     OVER
    ord_cboCustnum.AddString GIVING RESULT:
                USING CONAME
    ord_cboCustnum.SetItemData USING NWORK10:
                RESULT
    REPEAT
.
    READ      CUSTFILE;CUSTNUM;;
    RETURN
```

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

10. Add the call to OrdCust to the OrdOpen routine. This will fill the customer ComboBox with the current customers.

```

LISTINS   ORDCMD,ord_cmdSave,ord_cmdDelete:
          ord_cmdFirst,ord_cmdPrevious:
          ord_cmdNext,ord_cmdLast
*
.Load the Customer ComboBox
.
          CALL      OrdCust
*
.Attempt to read the first record
.
          CALL      OrdFirst
          CALL      OrdCount
          RETURN

```

11. Change the OrdNew routine to only perform the DELETEITEM instruction on the order number field rather than the collection. We also need to clear the selected item in the customer and sales rep ComboBoxes and put the current date into the EditDateTime object as the default.

```

OrdNew
          SET        ADDING
          SETPROP    ORDTEXT,ENABLED=1
          SETPROP    ord_txtNumber,READONLY=0
          SETPROP    ord_cmdDelete,ENABLED=1
          SETPROP    ord_cmdNew,ENABLED=0
          SETPROP    ord_cmdSave,ENABLED=0
          SETFOCUS   ord_txtNumber
          DELETEITEM ord_txtNumber,0
          SETITEM    ord_cboCustnum,0,0
          SETITEM    ord_cboSalesRep,0,0
          CLOCK      TIMESTAMP,MSG
          SETITEM    ord_edtDate,0,MSG
          SETITEM    ord_rdoUPS,0,1
          RETURN

```



**EXERCISE – ENHANCING THE ORDERS FORM - continued**

12. The largest modifications come in the OrdGet and OrdPut routines. In OrdPut, we need to add code that will locate the matching customer number with GetItemData and then position the ComboBox to that name. We also must manually decode the shipment method for our radio buttons. A RadioGroup or an array of Radio buttons would simplify the code but neither is currently supported by the Form Designer.

OrdPut

```

IF          (ORDNUM > 0)
MOVE        ORDNUM,ORDKEY
ELSE
CLEAR       ORDKEY
ENDIF
SETITEM     ord_txtNumber,0,ORDKEY

.

ord_cboCustnum.GetCount Giving NWORK10
FOR         RESULT,"1",NWORK10
ord_cboCustnum.GetItemData GIVING VALUE:
            USING RESULT
REPEAT      WHILE (VALUE != ORDCUST)
ord_cboCustnum.SetCurSel USING RESULT
SETITEM     ord_cboCustnum,0,ORDCUST

.

SETITEM     ord_edtDate,0,ORDDATE

.

IF          (ORDSHIP = 1)
SETITEM     ord_rdoUPS,0,1
ELSEIF      (ORDSHIP = 2)
SETITEM     ord_rdoFedEx,0,1
ELSE
SETITEM     ord_rdoDHL,0,1
ENDIF

.

SETITEM     ord_cboSalesRep,0,ORDSALES
RETURN

```

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

13. Likewise, our OrdGet routine will need additional logic in support of the new objects.

OrdGet

```

GETITEM    ord_txtNumber,0,Result
IF         ZERO
CLEAR     ORDNUM
ELSE
GETITEM    ord_txtNumber,0,ORDKEY
MOVE      ORDKEY,ORDNUM
ENDIF

.

GETITEM    ord_cboCustnum,0,RESULT
DECR      RESULT
ord_cboCustnum.GetItemData GIVING ORDCUST:
          USING RESULT

.

GETITEM    ord_edtDate,0,ORDDATE

.

GETITEM    ord_rdoUPS,0,RESULT
IF         NOT ZERO
MOVE      "1",ORDSHIP
ELSE
GETITEM    ord_rdoFedEx,0,RESULT
IF         NOT ZERO
MOVE      "2",ORDSHIP
ELSE
MOVE      "3",ORDSHIP
ENDIF
ENDIF

.

GETITEM    ord_cboSalesRep,0,ORDSALES
RETURN

```

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

Before completing the modifications to order.pls, we must modify the OrdVerify routine to support the new items. Our code requires an order number, customer number, and sales rep's initials.

```
* .....
.
.Enable the Save button when the required fields are input
.
OrdVerify
    SETPROP      ord_cmdSave,ENABLED=$FALSE
    GETITEM      ord_txtNumber,0,RESULT
    RETURN       IF ZERO
    GETITEM      ord_cboCustnum,0,RESULT
    RETURN       IF ZERO
    GETITEM      ord_cboSalesRep,0,RESULT
    RETURN       IF ZERO
.
    SETPROP      ord_cmdSave,ENABLED=$TRUE
    RETURN
```

14. We can now return to the form and ensure that the required called are in the event procedures.

```
Load_frmOrder
    CALL          OrdOpen
    RETURN

Close_frmOrder
    CALL          OrdClose
    RETURN

Validate_ord_cboCustnum
    CALL          OrdVerify
    RETURN
```

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

```
Validate_ord_cboSalesRep
    CALL      OrdVerify
    RETURN
Click_ord_cmdClose
    CALL      OrdClose
    RETURN
Click_ord_cmdDelete
    CALL      OrdDelete
    RETURN
Click_ord_cmdFirst
    CALL      OrdFirst
    RETURN
Click_ord_cmdLast
    CALL      OrdLast
    RETURN
Click_ord_cmdNew
    CALL      OrdNew
    RETURN
Click_ord_cmdNext
    CALL      OrdNext
    RETURN
Click_ord_cmdPrevious
    CALL      OrdPrevious
    RETURN
Click_ord_cmdSave
    CALL      OrdSave
    RETURN
Validate_ord_txtNumber
    CALL      OrdVerify
    RETURN
```

**EXERCISE – ENHANCING THE ORDERS FORM - continued**

15. The final modification is to add a call to OrdCust to the CustSave and CustDelete routines in "customer.pls". The call will reload the order form customer ComboBox when any changes are made to the customer table.

```
ENDIF
```

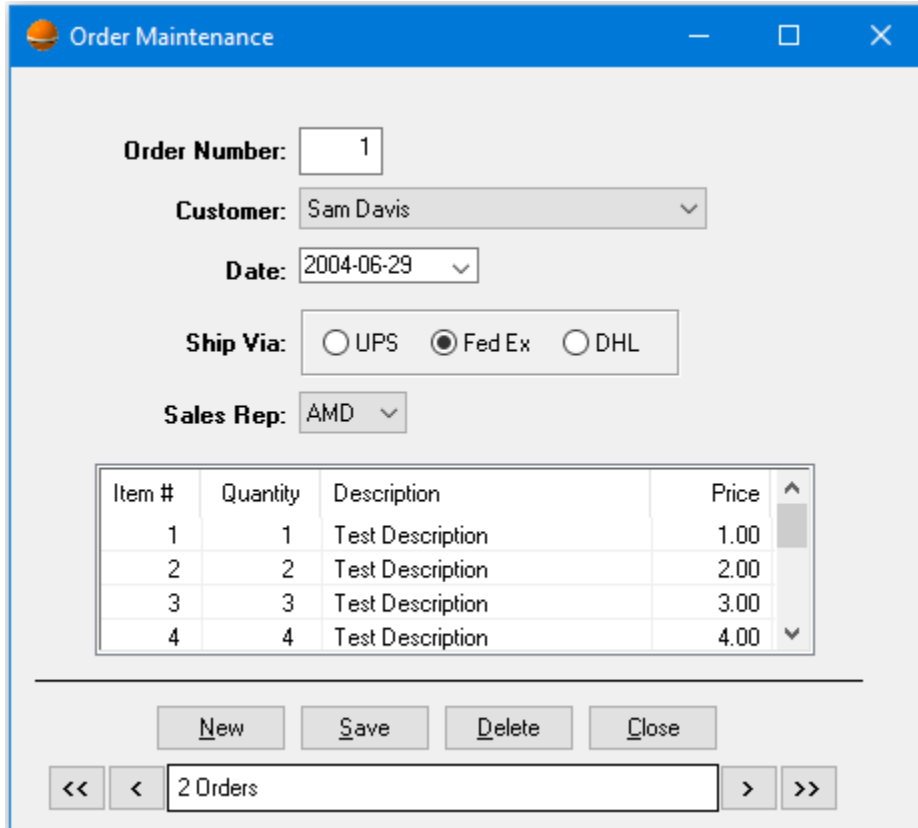
```
.
```

```
CALL      OrdCust  
RETURN
```

16. Compile and test your program.

## ADDING ORDER DETAIL

With the Order File maintenance complete, it is time to turn our attention to the Order Detail File. Our application will be designed to show the order's detail records in grid on the order form. Once we have the form in place and operational, we will add the ability to create, edit, or delete detail records.



The screenshot shows a Windows-style application window titled "Order Maintenance". The window contains several input fields and a table. The "Order Number" field contains the value "1". The "Customer" dropdown menu is set to "Sam Davis". The "Date" dropdown menu is set to "2004-06-29". The "Ship Via" section has three radio buttons: "UPS", "Fed Ex" (which is selected), and "DHL". The "Sales Rep" dropdown menu is set to "AMD". Below these fields is a table with four columns: "Item #", "Quantity", "Description", and "Price". The table contains four rows of data. At the bottom of the window, there are four buttons: "New", "Save", "Delete", and "Close". Below these buttons is a pagination bar with "<<" and "<" buttons on the left, a text box containing "2 Orders", and ">" and ">>" buttons on the right.

Item #	Quantity	Description	Price
1	1	Test Description	1.00
2	2	Test Description	2.00
3	3	Test Description	3.00
4	4	Test Description	4.00

**Order form showing Order Detail**

## ADDING ORDER DETAIL - continued

Let's begin by defining the data for the Order Details file. We will create a new source file named "detail.inc" and place the following definitions in it:

```
*
.Order Detail File
.
DTLORD      FORM      10
DTLITEM     FORM      10
DTLDESC     DIM        20
DTLQTY      FORM      10
DTLPRICE    FORM      7.2
.
DTLDATA     VARLIST    DTLORD, DTLITEM, DTLDESC, DTLQTY, DTLPRICE
.
DTLKEY      DIM        20
DTLFILE     IFILE
.
DTLTEXT     COLLECTION
```

This file is then included at the top of our main source file (sales.pls). Since we are using a ListView object to display detail items, we will go ahead and add the PLBMETH.INC inclusion file to the source file so we may use the special ListView defined constants.

```
INCLUDE     PLBEQU.INC
INCLUDE    PLBMETH.INC
INCLUDE     CUSTOMER.INC
INCLUDE     ORDER.INC
INCLUDE    DETAIL.INC
```

## ADDING ORDER DETAIL - continued

While modifying the main source file, we will need to add some new utility variables. These working variables are used during validation and retrieval of numeric data.

\*

.Utility Variables

.

RESULT	INTEGER	4
YESNO	INTEGER	1, "0x24"
ADDING	INTEGER	1
MSG	DIM	55
NWORK10	FORM	10
DIM10	DIM	10
<b>VALUE</b>	<b>INTEGER</b>	<b>4</b>
<b>SQTY</b>	<b>DIM</b>	<b>10</b>
<b>SPRICE</b>	<b>DIM</b>	<b>10</b>
<b>FORM72</b>	<b>FORM</b>	<b>7.2</b>
<b>TOP</b>	<b>FORM</b>	<b>4</b>
<b>LEFT</b>	<b>FORM</b>	<b>4</b>
ITEM	INTEGER	4

Finally, at the bottom of the main source we need to include the source file that will contain the order detail access routines.

```
INCLUDE    CUSTOMER.PLS
INCLUDE    ORDER.PLS
INCLUDE    DETAIL.PLS
```



## ADDING ORDER DETAIL - continued

Now let's modify the order form to include a ListView to display the detail data. The ListView is drawn beneath the order information and before the command and navigation buttons. A line is also drawn to visually separate the data area from the command and navigation tools.

The screenshot shows a Windows-style application window titled "Order Maintenance". The window has a blue title bar with standard minimize, maximize, and close buttons. The main area has a light gray background with a dotted grid pattern. The form contains the following elements:

- Order Number:** A text input field.
- Customer:** A dropdown menu.
- Date:** A date picker showing "2023-07-25".
- Ship Via:** A group box containing three radio buttons: "UPS" (selected), "Fed Ex", and "DHL".
- Sales Rep:** A dropdown menu.
- ListView:** A rectangular area with a dotted border and a light gray background, currently empty.
- Command Bar:** A horizontal bar at the bottom containing four buttons: "New", "Save", "Delete", and "Close".
- Navigation Bar:** A horizontal bar at the very bottom containing navigation arrows: "<<", "<", "Orders" (a text field), ">", and ">>".

We will name the ListView "ord\_lvDetail" and set the Sort Order property to "ascending".

## **ADDING ORDER DETAIL - continued**

With the ListView object in place, it is time to create the access routines for the order detail file. Use the IDE editor to create "detail.pls" and add the open routine as follows:

```
* .....
.
.Open the Order Detail File
.
DtlOpen
    TRAP      DtlPrep IF IO
    OPEN      DTLFILE,"ORDERDETAIL"
    TRAPCLR   IO
*
.Format the ListView
.
    ord_lvDetail.INSERTCOLUMN::
        USING "Item ##",50,0,LVCFMT_CENTER
    ord_lvDetail.INSERTCOLUMN::
        USING "Quantity",60,1,LVCFMT_CENTER
    ord_lvDetail.INSERTCOLUMN::
        USING "Description",165,2,LVCFMT_LEFT
    ord_lvDetail.INSERTCOLUMN::
        USING "Price",60,3,LVCFMT_RIGHT
.
    RETURN
```

The Open and ListIns logic is modeled after similar routines in Customer.pls and Order.pls. The last portion of the routine defines the columns of the ListView object, the column width, the index number and the column justification. The ListView alignment constants are defined in PLBMETH.INC we included previously.

**ADDING ORDER DETAIL - continued**

Next we need to add the Detail File creation logic. Again, this code is modeled after previously defined customer and order file routines. In this case however, we are going to add some temporary code to add detail items to order number one. This data is just some sample data so we will have something to display. We will remove the code in a later exercise.

```
* .....
.
.Create the Order Detail File
.
DtlPrep
    ALERT      TYPE=YESNO,"The Order Detail file "":
               "does not exist - Create It ?":
               RESULT,"Warning"
    STOP       IF (RESULT = 7)
.
    PREPARE    DTLFILE,"ORDERDETAIL","ORDERDETAIL":
               "1-20","60"
*
.Temporary Code
.
    FOR        RESULT,"1","10"
    MOVE       "1",DTLORD
    MOVE       RESULT,DTLITEM
    PACK       DTLDESC WITH "Test Description"
    MOVE       RESULT,DTLQTY
    MOVE       RESULT,DTLPRICE
    WRITE      DTLFILE;DTLDATA
    REPEAT
.
    RETURN
```

## ADDING ORDER DETAIL - continued

Finally, we will add a routine to read all detail records associated with the order and insert them into the ListView. The routine starts by removing any existing data. That is followed by a positional read and then a read key sequential loop. For each order detail record found, it is added to the ListView. Numeric data must be moved to string fields for insertion.

Note that we are also storing the item number in the row's PARAM field. The Item Number in the detail file is a sequential value that uniquely identifies the item. We don't want it to be visible to the user but we will need it to perform ISAM accesses. Our solution is to store the number in the PARAM field.

```
* .....
.
.Retrieve and Display Detail Records
.
DtllistAll
    ord_lvDetail.DeleteAllItems
    MOVE          ORDNUM,DTLKEY
    READ          DTLFILE,DTLKEY;;
.
    LOOP
    READKS        DTLFILE;DTLDATA
    UNTIL         (DTLORD != ORDNUM) OR OVER
    MOVE          DTLITEM,DTLKEY
    MOVE          DTLQTY,SQTY
    MOVE          DTLPRICE,SPRICE
    ord_lvDetail.InsertItemEx USING DTLKEY,Result:
        *Image=1,*Param=DTLITEM:
        *SubItem1=SQTY,*SubItem2=DTLDESC:
        *SubItem3=SPRICE
    REPEAT
    ord_lvDetail.GetItemCount Giving Result
    IF            ZERO
    ord_lvDetail.InsertItemEx USING "*",0
    ENDIF
    RETURN
```

## ADDING ORDER DETAIL - continued

The last step in attaching the detail records to our orders form is placing the calls to the new routines at the appropriate places in the orders access routine file (order.pls).

We begin by modifying the orders open routine to also open the detail file.

```
*
.Open the file
.
OrdOpen
        TRAP      OrdPrep IF IO
        OPEN      ORDFILE,"ORDER"
        TRAPCLR   IO
*
.Open the Order Detail file
.
        CALL      DtlOpen
```

Next, we add the call to the routine to show an order's detail records in the ListView at the bottom of the OrdPut routine:

```
.
        SETITEM   ord_cboSalesRep,0,ORDSALES
.
        CALL      DtlListAll
        RETURN
* .....
```

## **ADDING ORDER DETAIL - continued**

To prevent confusion, we should clear any detail records from the ListView when a new order is being added. Add the following line to the OrdNew routine:

```
        SETITEM    ord_rdoUPS,0,1
        ord_lvDetail.DeleteAllItems
    RETURN
* .....
```

If the user cancels a new order by clicking the Delete button or actually deletes an exiting order, we need to make sure the detail records correctly reflect the displayed order. At the bottom of the OrdDelete routine, add:

```
        ENDIF
.
        CALL      DtlListAll
    RETURN
* .....
```

## **EXERCISE – ADDING ORDER DETAIL**

In this exercise, we will add a ListView object and its supporting routines as described on the previous pages:

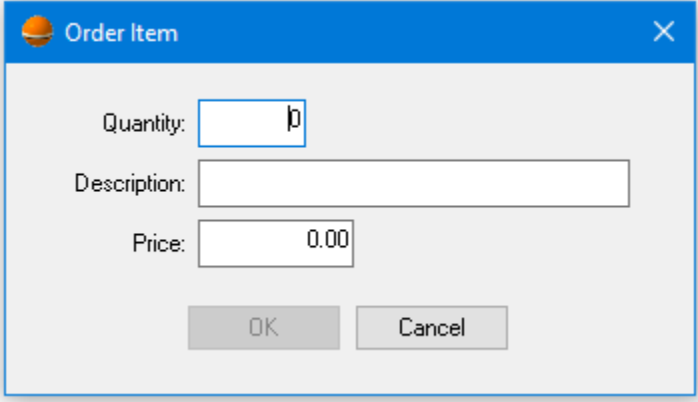
1. Create a source file containing the Order Detail file definitions and save it as "detail.inc".
2. Add "detail.inc" and "plbmeth.inc" inclusion files to the main source program (sales.pls).
3. Add the "value", "sqty", and "sprice" working variables to the main source program.
4. Include "detail.pls" at the bottom of the main source program.
5. Draw a ListView object on the order form. Place and size the object as previously illustrated. Name the object "ord\_lvDetail" and set the sort order property to "ascending".
6. Create a new source file (do not add to the project) named "detail.pls".
7. Add the previously described "DtIOpen", "DtIPrep", and "DtIListAll" routines to "details.pls".
8. Modify the "OrdOpen", "OrdPut", "OrdNew", and "OrdDelete" routines to call the detail file access routines as required.
9. Save all files and forms.
10. Compile and test the application.

## ORDER DETAIL MAINTENANCE

With our application now displaying detail records relating to the order, we need to add the ability to maintain these detail records. User will need to add, edit, and delete items on the order.

Our solution to this need will be to add a menu containing these functions when a right mouse click is performed on the ListView of detail records. If adding or editing a detail record a new Detail Item form is displayed. If deleting a detail item, it is simply removed from the file and from the ListView.

Let's begin by creating the Item Detail form:



**Item Maintenance Form**

This form is named "detail.plf". It consists of two EditNumber objects and an EditText object. The form's object prefix property is set to "dtl\_" and its caption is set to "Order Item". The Window Border property is set to "Fixed Single" and the Window Type property to "Modal Dialog". Set the form's Visible property to "False".



## **ORDER DETAIL MAINTENANCE - continued**

The first EditNumber object for the quantity field is named "dtl\_enQuantity" and its Alignment property is set to "Right". The integer digits property is set to "10" to indicate the maximum number of digits allowed.

An EditText object is used for the Description field. Its name is set to "dtl\_txtDescription" and its MaxChars property to "20".

The Price field also has an EditNumber object associated with it. Its name is "dtl\_enPrice" and its alignment is "Right". The object's Integer Digits is set to "7" and the Decimal Digits property to "2". These values match the item price data definition of a FORM 7.2.

Two command buttons are then placed on the form. The first button is named "dtl\_cmdOK" and it has a title property value of "OK". We will make it the default button by enabling the Default Property. We also want this button to be initially disabled so we set the Enabled property to "False".

The second command button is named "dtl\_cmdCancel" and has a title property of "Cancel". We set the button's Cancel property to True.

Before saving the form, ensure that the TabID properties are set to a meaningful sequence.

## ORDER DETAIL MAINTENANCE - continued

Within the code section of the form, we need to make the following calls for the defined object events.

```
Click_dtl_cmdCancel
    SETPROP      frmDetail,VISIBLE=$False
    RETURN
```

```
Click_dtl_cmdOK
    CALL         DtlSave
    RETURN
```

```
Validate_dtl_enPrice
    CALL         DtlVerify
    RETURN
```

```
Validate_dtl_enQuantity
    CALL         DtlVerify
    RETURN
```

```
Validate_dtl_txtDescription
    CALL         DtlVerify
    RETURN
```

The Order Item form is now complete. We need to include this form and load it in our main source program (sales.pls). Since the Order form (order.plf) refers to objects in the Detail form, the Detail form must be defined (included) before the Order form.

CUSTOMER	PLFORM	CUSTOMER.PLF
<b>DETAIL</b>	<b>PLFORM</b>	<b>DETAIL.PLF</b>
ORDER	PLFORM	ORDER.PLF
ABOUT	PLFORM	ABOUT.PLF
MAIN	PLFORM	MAIN.PLF
.		
	FORMLOAD	CUSTOMER
	FORMLOAD	ORDER
	<b>FORMLOAD</b>	<b>DETAIL</b>
	FORMLOAD	MAIN

## ORDER DETAIL MAINTENANCE - continued

Now we turn our attention to the Order Detail file access routines in detail.pls. We need to add a number of routines to this file.

We begin by modifying DtlOpen to construct a collection of the objects on the Order Item form:

```

        TRAP      DtlPrep IF IO
        OPEN      DTLFILE,"ORDERDETAIL"
        TRAPCLR   IO
*
.Build a collection of the input objects
.
        LISTINS   DTLTEXT,dtl_enQuantity:
                  dtl_txtDescription:
                  dtl_enPrice
*
.Format the ListView

```

We no longer need to generate sample data in the DtlPrep routine so we remove the code containing the FOR loop. We're left with:

```

DtlPrep
        ALERT     TYPE=YESNO,"The Order Detail file "":
                  "does not exist - Create It ?":
                  RESULT,"Warning"
        STOP      IF (RESULT = 7)
.
        PREPARE   DTLFILE,"ORDERDETAIL","ORDERDETAIL":
                  "1-20","60"
        RETURN

```

## ORDER DETAIL MAINTENANCE - continued

Next, we will add the standard routines that transfer the data between the file variables and the form objects:

```
* .....  
.  
.Retrieve Order Detail records from the ListView  
.  
DtlGet  
    GETITEM    dtl_enQuantity,0,SQTY  
    MOVE       SQTY,DTLQTY  
    GETITEM    dtl_txtDescription,0,DTLDESC  
    GETITEM    dtl_enPrice,0,SPRICE  
    MOVE       SPRICE,DTLPRICE  
    RETURN  
* .....  
.  
.Store Order Detail records into the ListView  
.  
DtlPut  
    MOVE       DTLQTY,SQTY  
    SETITEM    dtl_enQuantity,0,SQTY  
    SETITEM    dtl_txtDescription,0,DTLDESC  
    MOVE       DTLPRICE,SPRICE  
    SETITEM    dtl_enPrice,0,SPRICE  
    RETURN
```

## ORDER DETAIL MAINTENANCE - continued

We now move on to the data validation routine DtlVerify. We borrow logic from previously defined routines and enable the OK button only when all the required data is present.

```
* .....  
.  
.Enable the OK button when the required files are input  
.  
DtlVerify  
    DEBUG  
    SETPROP    dtl_cmdOK,ENABLED=$FALSE  
    GETITEM    dtl_enQuantity,0,Msg  
    MOVE       Msg,RESULT  
    RETURN     IF ZERO  
    GETITEM    dtl_txtDescription,0,RESULT  
    RETURN     IF ZERO  
    GETITEM    dtl_enPrice,0,MSG  
    MOVE       MSG,FORM72  
    RETURN     IF ZERO  
.  
    SETPROP    dtl_cmdOK,ENABLED=$TRUE  
    RETURN
```

## **ORDER DETAIL MAINTENANCE - continued**

A DtlNew routine is next added. It sets a flag indicating we are in add mode, empties the EditNumber and EditText objects, places the focus on the Quantity field, and makes the form visible.

```
* .....  
.  
.Add a Detail record for the current order  
.  
DtlNew  
    SET          Adding  
    DELETEITEM  DtlText,0  
    SETFOCUS    dtl_enQuantity  
    SETPROP     frmDetail,VISIBLE=$True  
    RETURN
```

**ORDER DETAIL MAINTENANCE - continued**

The DtlSave routine is responsible for writing new detail records or updating existing ones. If we are in addition mode, we begin by determining the next unused item number. This is done by positioning to the next order and reading in a reverse direction. If we find no detail records for the current order, the item number is set to one. Otherwise, we set the item number to item number read plus one.

```

* .....
.
.Save a Detail record for the current order
.
DtlSave
    IF          (Adding)
    CLEAR      RESULT
    ord_lvDetail.GetItemText Giving DTLKEY::
                USING Result
    IF          (DTLKEY = "*")
    ord_lvDetail.DeleteItem USING RESULT
    ENDIF
    MOVE        ORDNUM,DTLORD
    INCR        DTLORD
    MOVE        DTLORD,DTLKEY
    READ        DTLFILE,DTLKEY;;
    READKP      DTLFILE;NWORK10,DTLITEM
    IF          (NWORK10 = DTLORD)
    INCR        DTLITEM
    ELSE
    MOVE        "1",DTLITEM
    ENDIF
    CALL        DtlGet
    MOVE        ORDNUM,DTLORD
    WRITE        DTLFILE;DTLDATA
    MOVE        DTLITEM,DTLKEY
    ord_lvDetail.InsertItemEx USING DTLKEY,Result:
                *Param=DTLITEM,*SubItem1=SQTY:
                *SubItem2=DTLDESC,*SubItem3=SPRICE
    CLEAR      Adding
    ELSE

```

**ORDER DETAIL MAINTENANCE - continued**

DtlSave (continued).

```
CALL      DtlGet
UPDATE    DTLFILE;DTLDATA
MOVE      DTLQTY,SQTY
MOVE      DTLPRICE,SPRICE
ord_lvDetail.SetItemText USING ITEM,SQTY,1
ord_lvDetail.SetItemText USING ITEM,DTLDESC,2
ord_lvDetail.SetItemText USING ITEM,SPRICE,3
ENDIF

.

SETPROP   frmDetail,VISIBLE=$False
RETURN
```



**ORDER DETAIL MAINTENANCE - continued**

The DtlEdit routine is called from the shortcut menu to change the selected item. The routine retrieves the detail record using the item number stored in the Param field of the row. The data is then placed in the Order Item form, the OK button enabled, focus set to the quantity field, and the form is made visible.

```
* .....
.
.Edit a Detail record in the current order
.
DtlEdit
    ord_lvDetail.GetItemParam GIVING NWORK10::
        USING ITEM
    RETURN    IF ZERO
.
    PACK      DTLKEY WITH ORDNUM,NWORK10
    READ      DTLFILE,DTLKEY;DTLDATA
    IF        OVER
    ALERT     STOP:
              "Error locating the detail record":
              Result,"Error"
    RETURN
    ENDIF
.
    CALL      DtlPut
    SETPROP   dtl_cmdOK,ENABLED=$TRUE
    SETFOCUS  dtl_enQuantity
    SETPROP   frmDetail,VISIBLE=$True
    RETURN
```

**ORDER DETAIL MAINTENANCE - continued**

The DtlDelete routine is called by the shortcut menu. It deletes the selected item from the file and removes it from the ListView.

```

* .....
.
.Delete a Detail record for the current order
.
DtlDelete
    ord_lvDetail.GetItemParam GIVING NWORK10::
        USING ITEM
    RETURN    IF ZERO
*
.Delete the record
.
    PACK      DTLKEY WITH ORDNUM,NWORK10
    DELETE    DTLFILE,DTLKEY
    IF        OVER
    ALERT     STOP:
              "Error deleting the detail Record.":
              RESULT,"Error"
    RETURN
    ENDIF
*
.Delete the Listview Item
.
    ord_lvDetail.DeleteItem USING ITEM
*
.If the ListView is empty, add the placeholder symbol
.
    ord_lvDetail.GetItemCount Giving Result
    IF        ZERO
    ord_lvDetail.InsertItemEx USING "*"
    ENDIF
.
    RETURN

```

**ORDER DETAIL MAINTENANCE - continued**

The DtlClick routine is called in response to a right mouse click on the ListView object. The routine is passed the row number as a parameter. That value is used to retrieve the position of the selected item. The position is used in turn to calculate the position of the shortcut menu that is then made visible.

```
* .....
.
.Show menu for right click on the detail listview
.
DTLCLICK  ROUTINE    Item
*
.Determine the click position
.
        ord_lvDetail.GetItemRect Giving Msg::
                Using Item,0
        SETLPTR    Msg,8
        RESET      Msg,5
        MOVE       Msg,Top
        ADD        "2",TOP
        SETLPTR    Msg,12
        RESET      MSG,9
        MOVE       MSG,Left
        ADD        "10",Left
        SETPROP    ord_mnuRight,TOP=Top,LEFT=Left
        ACTIVATE   ord_mnuRight
        RETURN
```

Finally, a small routine is added that is called when an item is selected on the shortcut menu. This routine receives the selected item number and branches to the correct routine.

```
* .....
.
.A right menu item has been selected
.
DtlMenu   ROUTINE    Value
          BRANCH     Value to DtlNew,DtlEdit,DtlDelete
          RETURN
```

## **ORDER DETAIL MAINTENANCE - continued**

The last file that needs modification to support the Order Detail access is the Order Form. We need to add the ability to show a FloatMenu when the right mouse button is clicked over the item number column of the detail ListView.

We start by opening the Order form using the Form Designer. A FloatMenu object is then drawn anywhere on the form. We next open the Form Outline window (F6) and expand the FloatMenu objects of "frmOrder".

The FloatMenu objects we just added needs to be named "ord\_mnuRight". We also must add three menu items as described below:

- |         |                                             |
|---------|---------------------------------------------|
| Item 1: | Name – mnu_ordRightNew<br>Text - &New       |
| Item 2: | Name – mnu_ordRightEdit<br>Text - &Edit     |
| Item 3: | Name – mnu_ordRightDelete<br>Text - &Delete |

## **ORDER DETAIL MAINTENANCE - continued**

Once the FloatMenu that will serve as our shortcut menu for order item accessing has been created and its properties set, we need to add code to handle the menu activation and selection.

To activate the menu when the right mouse button is clicked on the item column of the ListView, add the following call:

```
Click_ord_lvDetail
    CALL      DtlClick USING #EventResult::
              IF (#EventMod=16)
    RETURN
```

To dispatch a menu selection to the correct routine, add the following:

```
Click_ord_mnuRight
    CALL      DtlMenu USING #EventResult
    RETURN
```

## ORDER DETAIL MAINTENANCE – continued

Lastly, we need to tie some of the Order Item routines back to the Order file routines.

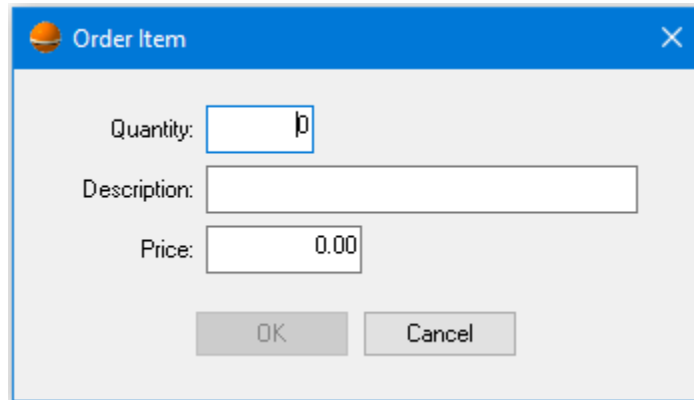
When an order is deleted, we must also delete all the associated item records. We do this by adding a CALL to the DtlDeleteAll routine to the OrdDelete routine.

```
OrdDelete
    IF          (Adding)
    CLEAR      ORDDATA,Adding
    SETPROP    ord_cmdNew,ENABLED=1
    SETPROP    ord_txtNumber,READONLY=1
    CALL       OrdPut
    CALL       OrdRead
    ELSE
    RETURN     IF (ORDNUM = 0)
    DELETE     ORDFILE
    CALL       DtlDeleteAll
    CALL       OrdNext
    CALL       OrdPrevious IF OVER
    CALL       OrdCount
    ENDIF
```

## **EXERCISE – COMPLETING THE ORDER DETAIL MAINTENANCE**

In this exercise, we will complete the Order Detail file maintenance.

1. Begin by designing the Order Item form described previously. This form contains three StatText objects, two EditNumber objects, an EditText object, and two Buttons.



The screenshot shows a dialog box titled "Order Item". It contains three input fields: "Quantity" with the value "10", "Description" which is empty, and "Price" with the value "0.00". At the bottom are "OK" and "Cancel" buttons.

2. Within the form, add the code statements to call the Order Detail access routines as appropriate.
3. Modify the main source file (sales.pls) to include and load the new form.
4. Modify or add the required new routines to the detail.pls inclusion file.
5. Add a FloatMenu to the Order form (order.plf) that supports the New, Edit, and Delete items. Code the menu's click event to call the DtlMenu access routine.
6. Code the ListView object to display the menu when the right button is clicked on the item number column.
7. Add a ToolTip message to the Listview that reads "Right click item number to edit".
8. Compile and test the application.

## **CHAPTER THIRTEEN**

### ***AAM DATA ACCESS***



**CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on the implementation of the Associative Access Method (AAM) in a Visual PL/B application.

Upon completion of this chapter, you will be able to:

- Use the Find button to locate a customer by any field.
- Implement additional Simplified File IO using the FILELIST statement.

## **THE CUSTOMER MAINTENANCE FORM**

You completed the customer maintenance form in the previous chapter for the Customer file. While the form is functional, its primary shortcoming is that a customer is located by scrolling through the entire customer file. For a small set of records, this is not an issue. But, as the customer file grows, the application must have a more efficient way of locating a particular customer record.

There are several approaches we could take to solve this problem. It would be a simple matter to add a Find button that invokes a form requesting the customer number. Once a value was entered, an ISAM read could retrieve the data that would then be displayed. The problem with this approach is that it assumes the user will always know the customer number.

A more sophisticated approach would be to allow the user to enter any of the customer file information. The program could then perform an AAM read to produce a subset of the customer records and navigate through this subset to locate the correct record. This solution allows the user to locate a customer using whatever data he has available. Additionally, text fields such as name, address, and contact names could implement a free-form search allowing specification of a value anywhere within the field. For example, you might know that the customer's last name is "White" and they are located in the city of "Tyler".

Implementing AAM access once meant that some carefully crafted code was required to keep the ISAM, AAM and data files synchronized. Sunbelt's Simplified File IO feature eliminates the need for that code. The implementation that follows uses the FILELIST instruction to manage the files.

## THE CUSTOMER DEFINITION FILE

The first step in adding AAM access to the sales application is to modify the customer definition file. The file modifications include the:

- definition of the number of fields in the customer file (8).
- initialization of each field's search mode
- creation of an array of keys. One array element per field is created. The size is the largest field size (40) plus three bytes for the AAM key overhead.
- definition of the customer AFILE used for AAM lookups and navigation.
- construction of the customer FILELIST that simplifies coding.

```
.
CUSTDATA  VARLIST      CUSTNUM, CONAME, ADDRESS, CITY:
                        STATE, ZIPCODE, CONTACTFN, CONTACTLN

.
CUSTKEY    DIM          10
CUSTFCNT   CONST       "8"
CUSTKEYA   DIM          43 (CUSTFCNT)
CUSTSRCH   DIM          1 (CUSTFCNT) , ("X") , ("F") , ("F") :
                        ("F") , ("X") , ("X") , ("F") , ("F")

.
CUSTLIST   FILELIST
CUSTFILE   IFILE       NAME="CUSTOMER.ISI"
CUSTFILA   AFILE       NAME="CUSTOMER.AAM"
                        FILELISTEND

.
CUSTTEXT   COLLECTION
CUSTCMD    COLLECTION
```

## THE MAIN PROGRAM

The addition of the AAM search function will require some additional work variables in main source file (sales.pls).

```
*
.Utility Variables
.
RESULT      INTEGER    4
YESNO       INTEGER    1,"0x24"
ADDING      INTEGER    1
MSG         DIM        55
NWORK10     FORM       10
DIM10       DIM        10
VALUE       INTEGER    4
SQTY        DIM        10
SPRICE      DIM        10
FORM72      FORM       7.2
TOP         FORM       4
LEFT        FORM       4
ITEM        INTEGER    4
FIELDNO    FORM       2
SEARCHING INTEGER    1
```

## THE CUSTOMER FORM

Once the customer definition file has been modified, we turn our attention to the customer form. A single button is added to the form. It is named "cust\_cmdFind" and the title property is set to "Find".

The code for the Find button's click event is:

```
Click_cust_cmdFind
    CALL      CustFind
    RETURN
```

**CUSTOMER ACCESS ROUTINES**

The routines encapsulated in customer.pls require the most modification. We will be adding logic to support both the maintenance and navigation of the AAM file.

We begin by modifying the CustOpen routine. Rather than just opening the ISAM index, we open the newly created FILELIST. This actually opens both the ISAM and AAM file.

```
*
.Open the file
.
CustOpen
        TRAP      CustPrep IF IO
        OPEN      CUSTLIST
        TRAPCLR   IO
```

We then move to the CustPrep routine. Here we simply need to add a PREPARE statement for the AAM file we're adding:

```
.
        PREPARE    CUSTFILE,"CUSTOMER","CUSTOMER":
                  "1-10","167"
        PREPARE    CUSTFILA,"CUSTOMER","CUSTOMER":
                  "U,1-10,11-50,51-90,91-120,"::
                  "121-122,123-127,128-147,148-167","167"
        RETURN
```

**CUSTOMER ACCESS ROUTINES - continued**

The CustClose routine is modified to serve two purposes. When the user is not searching for records, the routine will simply hide the customer form as before. If we have a search in process, the Close button will be re-labeled "Cancel". In this state, clicking the button will terminate the search mode. The button will then be labeled "Close" again and it will resume normal operation.

CustClose

```
IF          (Searching)
  SETPROP   CUSTCMD,ENABLED=1
  CLEAR     CUSTDATA,Searching
  SETPROP   cust_cmdNew,ENABLED=1
  SETPROP   cust_txtNumber,READONLY=1
  SETPROP   cust_cmdFind,TITLE="&Find"
  SETPROP   cust_cmdDelete,TITLE="&Delete"
  CALL      CustPut
  CALL      CustRead
  CALL      CustCount
ELSE
  CALL      CustDelete IF (Adding)
  SETPROP   frmCustomer,VISIBLE=0
ENDIF
RETURN
```

**CUSTOMER ACCESS ROUTINES - continued**

The CustFirst and CustLast routines are modified to position to the first record in the set of selected AAM records or the last record in the set if a search is in progress. Otherwise, the routines perform as they did previously:

CustFirst

```

IF                (SEARCHING)
READ              CUSTFILA,CUSTKEYA;CUSTDATA
ELSE
  FILL            " ",CUSTKEY
  READ            CUSTFILE,CUSTKEY;;
  READKS          CUSTFILE;CUSTDATA
.
  IF              OVER
  SETPROP          CUSTTEXT,ENABLED=0
  SETPROP          CUSTCMD,ENABLED=0
  RETURN
  ENDIF
ENDIF
.
  CALL            CustPut
  RETURN
.
```

CustLast

```

IF                (SEARCHING)
READLAST          CUSTFILA,CUSTKEYA;CUSTDATA
ELSE
  FILL            "9",CUSTKEY
  READ            CUSTFILE,CUSTKEY;;
  READKP          CUSTFILE;CUSTDATA
  RETURN          IF OVER
  ENDIF
.
  CALL            CustPut
  RETURN
.
```



## CUSTOMER ACCESS ROUTINES - continued

The only modification to the CustDelete routine is to replace the ISAM DELETE instruction with a FILELIST DELETE.

```
ELSE
RETURN      IF (CUSTNUM = 0)
DELETE     CUSTLIST
CALL       CustNext
```

In a similar fashion, the CustSave routine is modified to make use of the FILELIST logic:

```
CustSave
CALL      CUSTGET
.
IF        (ADDING)
WRITE    CUSTLIST;CUSTDATA
CALL      CustCount
CLEAR     ADDING
SETPROP   CUSTCMD,ENABLED=1
SETPROP   cust_cmdNew,ENABLED=1
SETPROP   cust_txtNumber,READONLY=1
ELSE
UPDATE   CUSTLIST;CUSTDATA
ENDIF
.
SETPROP   cust_cmdSave,ENABLED=0
CALL      OrdCust
RETURN
```

**CUSTOMER ACCESS ROUTINES - continued**

The CustCount routine normally shows the number of customers on file in the navigation bar at the bottom of the form. When locating customers via the Find button, we are dealing with a subset of the customer file. It would be misleading to show the full customer count during this process. Since there is no easy way of knowing how many records are in the subset, we will add logic to CustCount to just output a generic message while searching.

```

* .....
.
.Update the count of Customers
.
CustCount
    IF          (SEARCHING)
    SETITEM     cust_lblCount,0,"Search Results"
    ELSE
    GETFILE     CUSTFILE,RECORDCOUNT=NWORK10
.
    IF          (NWORK10 = 0)
    MOVE        "No Customers",MSG
    ELSEIF      (NWORK10 = 1)
    MOVE        "1 Customer",MSG
    ELSE
    MOVE        NWORK10,DIM10
    SQUEEZE     DIM10,DIM10
    PACK        MSG WITH DIM10," Customers"
    ENDIF
.
    SETITEM     cust_lblCount,0,MSG
.
    IF          (NWORK10 > 1)
    SETPROP     CUSTCMD,ENABLED=$TRUE
    ENDIF
    ENDIF
    RETURN

```

**CUSTOMER ACCESS ROUTINES - continued**

The last modification is the addition of the CustFind routine that is attached to the Find button. This routine serves two purposes:

1. It prompts the user for the search criteria.
2. It performs the initial AAM read.

Upon initial entry, the routine sets a flag indicating the program is in search mode. The navigation, New, and Delete buttons are disabled. All of the field EditText objects are then enabled and their contents removed. The Close button is labeled "Cancel" to indicate its new functionality and the Find button is labeled "Search". The form is then ready for customer input. The values entered are the search criteria that will be used in the next phase of the routine.

Once the form is in search mode, the user has two options:

1. Click the Cancel button to return to normal mode.
2. Click Search to find the first matching record.

The real work of the routine occurs once the form is filled with the search criteria and the Search (Find) button is clicked. The routine begins by retrieving the values entered into the form by the user. The values are then placed into the AAM key array (CUSTKEYA) using some IMplode/EXplode logic. Each array element then has its field number and the AAM search mode character inserted into the beginning. As each key is built, a check is made to ensure that any field that should have a free-float search has the required minimum of three (3) characters.

Once the keys have been built, the initial AAM READ is done. If there are no matching records, an appropriate message is displayed. Otherwise, the first record is displayed and the navigation and control keys are enabled.

The routine ends by calling CustCount to update the message in the navigation section appropriately.

**CUSTOMER ACCESS ROUTINES,- continued**

```

*.....
.
.Locate a customer
.
CustFind
    IF          (Searching)
    CALL        CUSTGET
    CLEAR       CUSTKEYA
    IMplode     MSG,";",cust_txtNumber:
                cust_txtName,cust_txtAddress:
                cust_txtCity,cust_txtState:
                cust_txtZipcode,cust_txtFName:
                cust_txtLName
    EXplode     MSG,";",CUSTKEYA
    FOR         FIELDNO,"1",CUSTFCNT
    COUNT       RESULT,CUSTKEYA(FIELDNO)
    IF          NOT ZERO
    IF          (CUSTSRCH(FIELDNO)="F" AND RESULT<3)
    ALERT       CAUTION,"At least three characters "":
                "required for search":
                RESULT,"Find"

    RETURN
    ENDIF
    PACK        DIM10 WITH FIELDNO,CUSTSRCH(FIELDNO)
    REP         " 0",DIM10
    SPLICE      DIM10,CUSTKEYA(FIELDNO)
    ENDIF
    REPEAT
    READ        CUSTFILA,CUSTKEYA;CUSTDATA
    IF          OVER
    ALERT       NOTE,"No matching records gound":
                RESULT,"Find"

    SETFOCUS    cust_txtNumber
    RETURN
    ENDIF

```

**CUSTOMER ACCESS ROUTINES - continued**

## CustFind (continued)

```
CALL      CUSTPUT
SETPROP   cust_cmdDelete,ENABLED=1
SETPROP   cust_cmdFirst,ENABLED=1
SETPROP   cust_cmdPrevious,ENABLED=1
SETPROP   cust_cmdNext,ENABLED=1
SETPROP   cust_cmdLast,ENABLED=1
RETURN

ELSE
SET        SEARCHING
SETPROP    CUSTCMD,ENABLED=0
SETPROP    cust_cmdNew,ENABLED=0
SETPROP    cust_cmdDelete,ENABLED=0
SETPROP    CUSTTEXT,ENABLED=1
SETPROP    cust_txtNumber,READONLY=0
SETPROP    cust_cmdClose,ENABLED=1,Title="Cancel"
SETPROP    cust_cmdFind,Title="Search"
SETFOCUS   cust_txtNumber
DELETEITEM CUSTTEXT,0
ENDIF

CALL      CustCount
RETURN
```

**EXERCISE – ADDING AAM ACCESS**

Modify the customer form, definition file, and access file as described on the previous pages to incorporate an AAM lookup function.

1. Begin by modifying the customer definition file (customer.inc) to define the AAM keys, search type array, FILELIST, and AFILE definition.
2. Modify the customer form to include the Find button. Add logic to the button's click event to call the CustFind routine.
3. Add work variables as required to the main source program (sales.pls).
4. Modify each of the customer access routines (customer.pls) to support the search logic.
5. Add the CustFind routine to the customer access file.
6. Save all forms and code.
7. Should you wish to test the application and maintain the existing data, it is necessary to create the customer Aamdex file manually. Please enter the following command from a DOS window. The command should be entered on a single line.

```
\sunbelt\code\sunaamd\ CUSTOMER -U,1-10,11-50,51-90,91-120,  
121-122,123-127,128-147,148-167
```

8. Test the application.

## **CHAPTER FOURTEEN**

### ***ADVANCED PRINTING***





**CHAPTER OVERVIEW AND OBJECTIVES**

This chapter focuses on how to use the advanced printing verbs in Visual PL/B to generate reports that may be viewed or printed.

Upon completion of this chapter, you will be able to:

- Define the verbs that make up advanced printing.
- Demonstrate how to use advanced printing instructions to preview and print reports.
- Use list controls to produce unique printing effects.

## **INTRODUCTION TO ADVANCED PRINTING**

Visual PL/B supports new instructions that allow printer output to be generated in much the same way forms are created. Rather than being bound to output lines from the top of the page to the bottom, the programmer places data or images on the page in any sequence. Advanced printing supports multiple job streams from the same program, a print preview function, and better control using options.

Advanced printing instructions employ a PFILE data item. The PFILE data item is similar in definition and uses a file handle and the disk IO instructions. A PFILE is associated with the actual output device by the PRTOPEN instruction. Subsequent advanced printing instructions all reference the device using these definitions. Multiple PFILE may be defined and open at any given time.

The syntax of a PFILE definition is simply:

```
[label]    PFILE
```

Advanced printing consists of the following set of instructions:

<b><u>Instruction</u></b>	<b><u>Use</u></b>
PRTOPEN	Associates the PFILE definition with an output device
PRTPAGE	Outputs text and control string to the print device
PRTPLAY	Output a previously generated file to a print device
PRTCLOSE	Terminates use of any output device

## **PRTOPEN**

The first step in generating advanced print output is to open the output device. This step associates the device with a previously defined PFILE data item. The syntax of the PRTOPEN statement is:

```
PRTOPEN [{window};]{pfile},{device}:  
        {jobname}[,{options}...]
```

Window is an optional previously created WINDOW object variable.

Pfile is a required PFILE variable.

Device is the required name of a Windows print device. If a null device name is specified, the standard Windows print dialog will be displayed.

Jobname is the required string that identifies the job for the Windows print manager. If a null job name is specified, the Windows print manager will list the job as "untitled".

**PRTOPEN - continued**

Options are one or more of the keywords listed below:

**Option****Specifies ...**

CANCELKEY={nvar|ivar}

a numeric variable that is assigned a value of one if the print dialog was cancelled. Note that an S10 error will still occur if the dialog is cancelled. If the user program TRAPs the S10 error, the CANCELKEY variable value is set accordingly.

PAGENUM={value}

a value that controls the page number edit fields in a PRINT dialog. If the {value} is zero, the page number edit fields are disabled. If the {value} is not zero, the page number edit fields are enabled. When this option is not specified, the page number edit fields are enabled. Use the GETINFO instruction to retrieve print dialog user responses.

PAGEWIDTH={value}

a numeric variable that receives the page width calculated using the character width of the default font.

PAGEHEIGHT={value}

a numeric variable that receives the page height calculated using the character width of the default font.

PIXELWIDTH={value}

a numeric variable that receives the pixel width of the page.

PIXELHEIGHT={value}

a numeric variable that receives the pixel height of the page.

**PRTOPEN – continued**

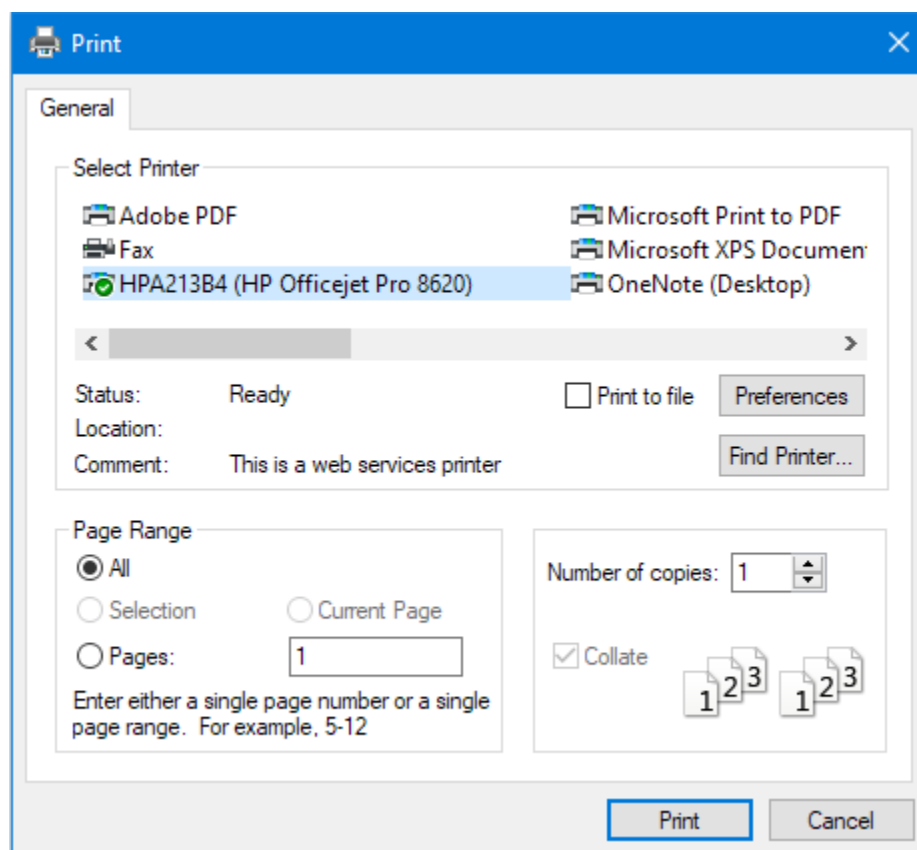
MAXPAGE={value}	a numeric variable or decimal constant indicating the initial value for the maximum number of pages entry in the print dialog.
NOPRINT	that data should be spooled only. The PRINT button is disabled.
RETSPLNAME={value}	a string to receive the qualified spool file name after the instruction is completed. If the spool file option is not specified, the RETSPLNAME option will null the string.
SELECTION={value}	a numeric variable or decimal constant that controls the print dialog selection field. If the {value} is zero, the selection control is disabled. If the {value} is not zero, the selection control is enabled. When this option is not specified, the selection field is disabled. Use the GETINFO instruction to retrieve print dialog user responses.
spool file={value}	a character variable or literal indicating the spool file name. The default spool extension is acquired from screen definition file. If {value} is a Null String, a standard Windows file prepare dialog is invoked.

## PRTOPEN – continued

Example:

```
P1          PFILE
           PRTOPEN  P1, "", ""
```

This example opens the Windows print dialog and allows the user to select a print device. The job name defaults to "untitled".



Windows Print Dialog

## **PRTCLOSE**

The PRTCLOSE instruction terminates printing to a Windows print device. The instruction uses the following format:

```
PRTCLOSE {pfile}[, {option}[, ...]]
```

Pfile is the required previously defined PFILE variable that has been opened using a PRTOPEN instruction.

Option is one of the keywords as described below that control the print preview initial presentation state.

After a PRTCLOSE, execution of a PRTPAGE instruction will cause an S05 runtime error.

If a device name of "@" or "@printer name" is specified in the PRTOPEN statement, a Print Preview is performed.

If Print Preview is not active when the PRTCLOSE instruction is executed, {option} is ignored.

**PRTCLOSE - continued**

Valid options may be of the following keywords that control the initial presentation state of the Print Preview:

**Option**

ABORT[={nvar|ivar}]

CANCELKEY={nvar|ivar}

COPYCOUNT={nvar|ivar}

DIALOG={value}

**Specifies ...**

that a user may terminate an advanced printing session without coming up into the PRINT PREVIEW window. Notice that the ABORT keyword can be specified with or without an optional parameter. When the {dnumnvar} value is specified, a non-zero value terminates the printing session while a value of zero does not terminate the printing session. In addition, note that any spooled file is deleted if a printing session is terminated.

a numeric variable that is assigned a value of one if the print dialog was cancelled. Note that an S10 error will still occur if the dialog is cancelled. If the user program TRAPs the S10 error, the CANCELKEY variable value is set accordingly.

a numeric variable that identifies the current setting of the printer driver copy parameter when a printing operation was performed.

whether a print dialog should be presented when a Print is selected from a Print Preview operation. When the {value} is zero, a print dialog does not appear. When the {value} is not zero, a print dialog does appear. When this option is not specified, the default action is a {value} of zero.



**PRTCLOSE - continued**

ENDPAGE={value}	the ending physical page is the last page in the Print Preview presentation. If this option is not specified, the last page defaults to the last page generated by a PRTPAGE operation before the PRTCLOSE operation.
GOTOPAGE={value}	the first page presented from the print preview page set being used. The {value} should be between one and the last logical page being reviewed.
MAXIMIZE={value}	whether the Print Preview window starts in a maximized state. A {value} of zero (0) indicates that the window is not maximized when started. This is also the default starting state. Any non-zero {value} indicates that the window should start in a maximized state.
PAGECOUNT={nvar ivar}	a numeric variable that identifies the number of pages printed. If the value returned is zero, no pages were printed. This keyword indicates whether a user printed from a Print Preview window.
STARTPAGE={value}	the beginning physical page is page one in the Print Preview presentation. If this option is not specified, page one defaults to the first physical page generated after the PRTOPEN operation.
ZOOM={value}	the initial zoomed state for the Print Preview window. {value} can be in the range of zero (0) to six (6) inclusive. A {value} of zero indicates that zooming should start at the minimum size. This is the default starting zoom state when this keyword is not provided. A {value} of six indicates that zooming should start at the maximum size.

**PRTPAGE**

The workhorse of the Advanced Printing instructions is the PRTPAGE verb. This instruction outputs to a print device in page mode. Information may be placed anywhere on the page in any order. The instruction uses the following format:

```
PRTPAGE    {pfile};{list}[:,]
```

Pfile is the required previously defined PFILE variable that has been opened using a PRTOPEN instruction.

List is a list containing any combination of character string Literals, Character String Variables, numeric literals or variables (FORM), VARLISTs of DIM, INIT or FORM variables, ARRAYs, individual array elements, valid control characters, and valid List Controls,

An optional trailing semicolon inhibits the normal carriage return and line feed.

List controls format the printing of the data. The following List Controls are supported:

*ALIGNMENT	*FGCOLOR	*OVERLAYON	*RNDRECT
*ALLOFF	*FILL	*P	*ROWSPACE
*BGCOLOR	*FONT	*PAPER	*RPTCHAR
*BIN	*H	*PENSIZE	*RPTDOWN
*BLANKOFF	*HA	*PICT	*SL
*BLANKON	*JC	*PICTCLIP	*TAB
*BOLDOFF	*MARGINL	*PICTRECT	*ULOFF
*BOLDON	*MARGINR	*PICTVIS	*ULON
*C	*L	*PIXELOFF	*UNITS
*COLLATE	*LINE	*PIXELON	*V
*COLOROFF	*LL	*PL	*VA
*COLORUSE	*N	*PLENGTH	*ZF
*COLSPACE	*NEWPAGE	*PSCALE	*ZFOFF
*COPIES	*ORIENT	*PWIDTH	*ZFON
*DUPLEX	*OVAL	*QUALITY	*ZS
*F	*OVERLAYOFF	*RECT	

**PRTPAGE EXAMPLE**

```

P          PFILE
TITLE      INIT      "PRTPAGE Sample Output"
DATE       DIM        8
FORM1      FORM      "98.654"
*
.Allow the User to Select a Printer
.
          PRTOPEN    P, "", ""
*
.Print an Oval and Three Round Rectangles
.
          CLOCK      DATE, DATE
          PRTPAGE    P; *BGCOLOR=*YELLOW: ;Gray fill
                   *P50:1, TITLE: ;Output the Title
                   *P56:2, DATE: ; and the Date
                   *FILL=*ON: ;Set fill mode on
                   *OVAL=2:10:10:40: ;Print oval object
                   *FILL=*OFF: ;Set fill mode off
                   *PENSize=10: ;Change width of pen
                   *RNDRECT=12:20:10:40:1:1: ;Round rectangle
                   *RNDRECT=22:30:10:40:2:2: ;Round rectangle
                   *RNDRECT=32:40:10:40:3:3: ;Round rectangle
                   *BGCOLOR=*WHITE;
*
.Print a Filled Rectangle
.
          PRTPAGE    P; *BGCOLOR=*YELLOW, *FILL=*ON:
                   *RECT=42:50:10:70, *OVERLAYON;
*
.Print a Field Using Each of the Alignment Properties
.
          PRTPAGE    P; *P40:52, *ALIGNMENT=*RIGHT, "Line 1":
                   *P40:53, "Line 2 ABCDEF":
                   *P40:54, *ALIGNMENT=*LEFT, "Line 3":
                   *ALIGNMENT=*DECIMAL:
                   *P40:55, "12345.234":
                   *P40:56, "23.45", *P40:57, FORM1
*
.Close the Printer and Exit
.
          PRTCLOSE    p

```

**This example prints an oval, three round rectangles, a filled rectangle, and aligned data to a user selected printer.**

**PRTPAGE EXAMPLE**

```
PF          PFILE

          . . . .

PRTOPEN    PF,"@?", ""

          . . . .

PRTCLOSE PF,ABORT          ;Aborts print session

PRTCLOSE PF,ABORT=1        ;Aborts print session

PRTCLOSE PF,ABORT=0        ;Does not abort

PRTCLOSE PF,ABORT=ABTFLG   ;Optional ABORT
```

Examples of PRTCLOSE.

**EXERCISE – ADVANCED PRINT**

Add an order report to the Sales application using Advanced Printing instructions. The report should:

- Produce results like the sample below.
- Be initiated from the File menu.
- Allow print previewing.



**Sales Application**  
**ORDER REPORT FOR 10-07-04**

Justin Sturgis ( 3  
 856 Riverside Drive North  
 Grapevine, TX 76750  
 Justin Sturgis

Order: 1 Date: 20040629 Ship Via2 Salesman:			
ITEM	DESCRIPTION	QUANTITY	PRICE
1	Test Description	1	1.00
2	Test Description	2	2.00
3	Test Description	3	3.00
4	Test Description	4	4.00
5	Test Description	5	5.00
6	Test Description	6	6.00
7	Test Description	7	7.00
8	Test Description	8	8.00
9	Test Description	9	9.00
10	Test Description	10	10.00
1	3	3	30.00

**APPENDIX A**

***CHARACTER TRANSLATION TABLE***





**TRANSLATION TABLE**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
0		000	00		NULL	NULL	0000 0000
1	(^A)	001	01	J	SOH	SOH	0000 0001
2	(^B)	002	02	Ä	STX	STX	0000 0010
3	(^C)	003	03	©	ETX	ETX	0000 0011
4	(^D)	004	04	¨	EOT	PF	0000 0100
5	(^E)	005	05	§	ENQ	HT	0000 0101
6	(^F)	006	06	ª	ACK	LC	0000 0110
7	(^G)	007	07	!	BEL	DEL	0000 0111
8	(^H)	010	08	ï	BS		0000 1000
9	(^I)	011	09	"	HT	RLF	0000 1001
10	(^J)	012	0A	Ð	LF	SMM	0000 1010
11	(^K)	013	0B	¿	VT	VT	0000 1011
12	(^L)	014	0C	À	FF	FF	0000 1100
13	(^M)	015	0D	Ç	CR	CR	0000 1101
14	(^N)	016	0E	È	SO	SO	0000 1110
15	(^O)	017	0F	Á	SI	SI	0000 1111
16	(^P)	020	10	'	DLE	DLE	0001 0000
17	(^Q)	021	11	(	DC1/XON	DC1	0001 0001
18	(^R)	022	12	&	DC2	DC2	0001 0010
19	(^S)	023	13	Ë	DC3/XOFF	DC3/TM	0001 0011
20	(^T)	024	14	Œ	DC4	RES	0001 0100
21	(^U)	025	15	§	NAK	NL	0001 0101
22	(^V)	026	16	É	SYN	BS	0001 0110
23	(^W)	027	17	Ì	ETB	IL	0001 0111
24	(^X)	030	18	–	CAN	CAN	0001 1000

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
25	(^Y)	031	19	—	EM	EM	0001 1001
26	(^Z)	032	1A	®	SUB	CC	0001 1010
27	(^[)	033	1B	¬	ESC	CU1	0001 1011
28	(^\)	034	1C	Î	FS	IFS	0001 1100
29	(^])	035	1D	«	GS	IGS	0001 1101
30	(^^)	036	1E	)	RS	IRS	0001 1110
31	(^_)	037	1F	Ú	US	IUS	0001 1111
32		040	20	SP	SPACE	DS	0010 0000
33		041	21	!	!	SOS	0010 0001
34		042	22	"	"	FS	0010 0010
35		043	23	#	#		0010 0011
36		044	24	\$	\$	BYP	0010 0100
37		045	25	%	%	LF	0010 0101
38		046	26	&	&	ETB	0010 0110
39		047	27	'	'	ESC	0010 0111
40		050	28	(	(		0010 1000
41		051	29	)	)		0010 1001
42		052	2A	*	*	SM	0010 1010
43		053	2B	+	+	CU2	0010 1011
44		054	2C	,	,		0010 1100
45		055	2D	-	-	ENQ	0010 1101
46		056	2E	.	.	ACK	0010 1110
47		057	2F	/	/	BEL	0010 1111
48		060	30	0	0		0011 0000

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
49		061	31	1	1		0011 0001
50		062	32	2	2	SYN	0011 0010
51		063	33	3	3		0011 0011
52		064	34	4	4	PN	0011 0100
53		065	35	5	5	RS	0011 0101
54		066	36	6	6	UC	0011 0110
55		067	37	7	7	EOT	0011 0111
56		070	38	8	8		0011 1000
57		071	39	9	9		0011 1001
58		072	3A	:	:		0011 1010
59		073	3B	;	;	CU3	0011 1011
60		074	3C	<	<	DC4	0011 1100
61		075	3D	=	=	NAK	0011 1101
62		076	3E	>	>		0011 1110
63		077	3F	?	?	SUB	0011 1111
64		100	40	@	@	SPACE	0100 0000
65		101	41	A	A		0100 0001
66		102	42	B	B		0100 0010
67		103	43	C	C		0100 0011
68		104	44	D	D		0100 0100
69		105	45	E	E		0100 0101
70		106	46	F	F		0100 0110
71		107	47	G	G		0100 0111
72		110	48	H	H		0100 1000

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
73		111	49	I	I		0100 1001
74		112	4A	J	J		0100 1010
75		113	4B	K	K	.	0100 1011
76		114	4C	L	L	<	0100 1100
77		115	4D	M	M	(	0100 1101
78		116	4E	N	N	+	0100 1110
79		117	4F	O	O		0100 1111
80		120	50	P	P	&	0101 0000
81		121	51	Q	Q		0101 0001
82		122	52	R	R		0101 0010
83		123	53	S	S		0101 0011
84		124	54	T	T		0101 0100
85		125	55	U	U		0101 0101
86		126	56	V	V		0101 0110
87		127	57	W	W		0101 0111
88		130	58	X	X		0101 1000
89		131	59	Y	Y		0101 1001
90		132	5A	Z	Z	!	0101 1010
91		133	5B	[	[	\$	0101 1011
92		134	5C	\	\	*	0101 1100
93		135	5D	]	]	)	0101 1101
94		136	5E	^	^	;	0101 1110
95		137	5F				0101 1111
96		140	60	`	`	-	0110 0000

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
97		141	61	a	a		0110 0001
98		142	62	b	b		0110 0010
99		143	63	c	c		0110 0011
100		144	64	d	d		0110 0100
101		145	65	e	e		0110 0101
102		146	66	f	f		0110 0110
103		147	67	g	g		0110 0111
104		150	68	h	h		0110 1000
105		151	69	i	i		0110 1001
106		152	6A	j	j		0110 1010
107		153	6B	k	k	,	0110 1011
108		154	6C	l	l	%	0110 1100
109		155	6D	m	m	_	0110 1101
110		156	6E	n	n	>	0110 1110
111		157	6F	o	o	?	0110 1111
112		160	70	p	p		0111 0000
113		161	71	q	q		0111 0001
114		162	72	r	r		0111 0010
115		163	73	s	s		0111 0011
116		164	74	t	t		0111 0100
117		165	75	u	u		0111 0101
118		166	76	v	v		0111 0110
119		167	77	w	w		0111 0111
120		170	78	x	x		0111 1000

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
121		171	79	y	y	/	0111 1001
122		172	7A	z	z	:	0111 1010
123		173	7B	{	{	#	0111 1011
124		174	7C			@	0111 1100
125		175	7D	}	}	'	0111 1101
126		176	7E	~	~	=	0111 1110
127		177	7F	DEL	DEL	"	0111 1111
128		200	80	Ç			1000 0000
129		201	81	ü		a	1000 0001
130		202	82	é		b	1000 0010
131		203	83	â		c	1000 0011
132		204	84	ä		d	1000 0100
133		205	85	à		e	1000 0101
134		206	86	å		f	1000 0110
135		207	87	ç		g	1000 0111
136		210	88	ê		h	1000 1000
137		211	89	ë		i	1000 1001
138		212	8A	è			1000 1010
139		213	8B	ï			1000 1011
140		214	8C	î			1000 1100
141		215	8D	ì			1000 1101
142		216	8E	Ä			1000 1110
143		217	8F	Å			1000 1111
144		220	90	É			1001 0000
145		221	91	æ		j	1001 0001
146		222	92	Æ		k	1001 0010

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
147		223	93	ô		l	1001 0011
148		224	94	ö		m	1001 0100
149		225	95	ò		n	1001 0101
150		226	96	û		o	1001 0110
151		227	97	ù		p	1001 0111
152		230	98	n		q	1001 1000
153		231	99	Ö		r	1001 1001
154		232	9A	Ü			1001 1010
155		233	9B	ç			1001 1011
156		234	9C	£			1001 1100
157		235	9D	¥			1001 1101
158		236	9E	&			1001 1110
159		237	9F				1001 1111
160		240	A0	á			1010 0000
161		241	A1	í		~	1010 0001
162		242	A2	ó		s	1010 0010
163		243	A3	ú		t	1010 0011
164		244	A4	ñ		u	1010 0100
165		245	A5	Ñ		v	1010 0101
166		246	A6	ª		w	1010 0110
167		247	A7	º		x	1010 0111
168		250	A8	¿		y	1010 1000
169		251	A9	Í		z	1010 1001
170		252	AA	¬			1010 1010

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
171		253	AB	$\frac{1}{2}$			1010 1011
172		254	AC	$\frac{1}{4}$			1010 1100
173		255	AD	i			1010 1101
174		256	AE	«			1010 1110
175		257	AF	»			1010 1111
176		260	B0				1011 0000
177		261	B1				1011 0001
178		262	B2				1011 0010
179		263	B3				1011 0011
180		264	B4				1011 0100
181		265	B5				1011 0101
182		266	B6				1011 0110
183		267	B7	+			1011 0111
184		270	B8	+			1011 1000
185		271	B9				1011 1001
186		272	BA				1011 1010
187		273	BB	+			1011 1011
188		274	BC	+			1011 1100
189		275	BD	+			1011 1101
190		276	BE	+			1011 1110
191		277	BF	+			1011 1111
192		300	C0	+			1100 0000
193		301	C1	-		A	1100 0001
194		302	C2	-		B	1100 0010



**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
195		303	C3	+		C	1100 0011
196		304	C4	-		D	1100 0100
197		305	C5	+		E	1100 0101
198		306	C6			F	1100 0110
199		307	C7			G	1100 0111
200		310	C8	+		H	1100 1000
201		311	C9	+		I	1100 1001
202		312	CA	-			1100 1010
203		313	CB	-			1100 1011
204		314	CC				1100 1100
205		315	CD	-			1100 1101
206		316	CE	+			1100 1110
207		317	CF	-			1100 1111
208		320	D0	-			1101 0000
209		321	D1	-		J	1101 0001
210		322	D2	-		K	1101 0010
211		323	D3	+		L	1101 0011
212		324	D4	+		M	1101 0100
213		325	D5	+		N	1101 0101
214		326	D6	+		O	1101 0110
215		327	D7	+		P	1101 0111
216		330	D8	+		Q	1101 1000
217		331	D9	+		R	1101 1001
218		332	DA	+			1101 1010

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
219		333	DB				1101 1011
220		334	DC	—			1101 1100
221		335	DD				1101 1101
222		336	DE				1101 1110
223		337	DF	—			1101 1111
224		340	E0	I			1110 0000
225		341	E1	ß			1110 0001
226		342	E2	,		S	1110 0010
227		343	E3	X		T	1110 0011
228		344	E4	;		U	1110 0100
229		345	E5	[		V	1110 0101
230		346	E6	T		W	1110 0110
231		347	E7	\		X	1110 0111
232		350	E8	>		Y	1110 1000
233		351	E9	1		Z	1110 1001
234		352	EA	A			1110 1010
235		353	EB	L			1110 1011
236		354	EC	¥			1110 1100
237		355	ED	f			1110 1101
238		356	EE	M			1110 1110
239		357	EF	Ç			1110 1111
240		360	F0	°		0	1111 0000
241		361	F1	±		1	1111 0001
242		362	F2	³		2	1111 0010

**TRANSLATION TABLE - continued**

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
243		363	F3	£		3	1111 0011
244		364	F4	ó		4	1111 0100
245		365	F5	õ		5	1111 0101
246		366	F6	¸		6	1111 0110
247		367	F7	»		7	1111 0111
248		370	F8	0		8	1111 1000
249		371	F9	+		9	1111 1001
250		372	FA	,			1111 1010
251		373	FB	Ö			1111 1011
252		374	FC	'			1111 1100
253		375	FD	²			1111 1101
254		376	FE	#			1111 1110
255		377	FF				1111 1111

## **APPENDIX B**

### ***PL/B UTILITIES***



**PL/B UTILITIES**

The PL/B Utilities allow manipulation of various types of files. These files include not only ASCII text files, but also Sunbelt's Indexed Sequential Access Method (ISAM) files, Associated Aamdex Method (AAM) files, object code files and others.

BLOKEDIT	Creates a new text file by merging data from other text files.
BUILD	Quick and easy way to create a small text file.
OBJMATCH	Compare two files byte for byte.
REFORMAT	Change the format of a text file.
SUNAAMDIX	Creates an Associated Access Method file
SUNDUMP	Display and modify any type of file on disk.
SUNINDEX	Creates an Indexed Sequential Access Method file
SUNLIST	List a text file to the CRT, printer or to a print formatted file.
SUNSCAN	View a text file using the arrow keys and PgUp and PgDn keys.
SUNSORT	Rearrange the records within a text file into another sequence.
TXTMATCH	Comparison of two text files.

## **BLOKEDIT**

The BLOKEDIT utility copies all or part(s) of one or more text files into a new text file.

Instructions may be entered from the keyboard or retrieved from a text file.

This utility is useful for rearranging source programs, extracting routines from various programs to create a new program or for rebuilding ASCII data files.

BLOKEDIT also provides the option to insert text from the keyboard.

The format for the BLOKEDIT command line is:

```
BLOKEDIT [{cmdfile}],{output} -[opts]
```

Instructions are entered to the BLOKEDIT utility while positioned on line 24 of the CRT or as retrieved from the {cmdfile}. Valid instructions are as follows:

<b>Entry</b>	<b>Meaning</b>
FILENAME	Search for specified input file.
n-m	Copy lines n through m from the input file and write these lines into the output file.
" (double quote)	Enter or exit text mode. Any lines entered between these are considered as input and are written to the output file. These lines start or terminate the text entry only if the "" is the first and only data on that line. Otherwise, it is considered an invalid command (if not in text entry mode) or is written to the output file.
* (asterisk)	End this BLOKEDIT. Write the EOF on the {output} file and return to the operating system.
. (period)	Comment line in command file.

## **BLOKEDIT - continued**

If {cmdfile} is specified, it must be an ASCII text format file that contains the instructions for this particular BLOKEDIT procedure.

If {cmdfile} is not specified, all instructions for this particular BLOKEDIT procedure must be entered from the terminal keyboard.

The assumed file type (extension) on both files is TXT.

If {output} already exists and {option} was not specified, the following prompt is displayed:

```
OUTPUT FILE ALREADY EXISTS.  OVERWRITE IT (Y/N) ?
```

Any response other than 'Y' aborts the BLOKEDIT utility.

A search of all search paths is performed in an attempt to locate the specified files. If {output} is not found, it is created in the current directory.

{option} is not required and if specified must be as follows:

Option	Meaning
C	Compressed output file.
N	Uncompressed output file (default).
O	Overwrite {output} if it already exists.
?	Display the BLOKEDIT help information.



**BUILD**

The BUILD utility creates ASCII text files or null files. The command line format for the BUILD utility is:

```
BUILD {output} -[{{endchar}}]
```

If {output} already exists, it is overwritten and no error message generated.

If no path specification is given for {output}, a path search is performed to determine if it already exists.

The assumed file type on {output} is TXT.

{{endchar}} signals the BUILD utility that this particular BUILD is complete. The character specified must be the one and only character on that line to terminate the command.

If {{endchar}} is not specified, a null entry terminates the BUILD.

## **OBJMATCH**

The OBJMATCH utility compares two files and displays any differences between them.

```
OBJMATCH {file1},{file2}
```

{file1} and {file2} may include directory information.

Every byte of each file is compared.

If a mismatch is found, both the hexadecimal and ASCII interpretations of the data from both files is shown.

## **REFORMAT**

The REFORMAT utility converts the internal disk format of text files. It also recovers lost disk space resulting from records deleted via AAM or ISAM. The command line for REFORMAT is:

```
REFORMAT [{input}],{output} [-{options}]
```

If {input} is not specified, the user is prompted for the input file(s). When all files have been entered, terminate the list by entering an asterisk (\*).

If an extension is not given for either {input} or {output}, TXT is assumed.

REFORMAT first attempts to locate the {input} file(s) in the current path. If not found, a search of all search paths occurs.

{output} is overwritten if it exists.

The {input} and {output} files must be different files.

**REFORMAT – continued**

{options} are as follows:

Option	Meaning
B{n}	Block the records in {output} {n} records per sector. The sector size is assumed 256 bytes.
C	The {output} file will be space compressed using the standard tab character (Hex 09). This is the default option.
E={xx}	Set the output End of Record Type to be {xx} where {xx} is one of CR, LF, CRLF, or LFCR.
K	Kill the input file(s) after completion of the REFORMAT.
L{n}	{output} records are to be written {n} bytes long.
P	Pack as many records into each 256 byte sector without crossing sector boundaries. The number of records in each sector may then be variable.
R	The {output} file is record compressed, but not space compressed.
S	Segment records longer than the specified length. Valid only if the 'L' option is also specified.
T	Truncate records longer than the specified length. Valid only if the 'L' option is also specified.
Z	Display the run time statistics including records in and out and start and stop times.
?	Display the REFORMAT help screen.

## SUNAAMDX

The SUNAAMD utility creates an Associated Access Method (AAM) key file using the specified key positions in the corresponding data file.

```
SUNAAMD {txt} [, {aam} [, {Lnnnn}]] - [{opts}] {keyspecs}  
SUNAAMD {aam2 } - {I | R}
```

If no extension is specified for {txt}, TXT is assumed.

If {aam} is not specified, the file name in {txt} is used with the extension AAM. If {aam} is specified without an extension, AAM is used.

If neither {txt} nor {aam} are located in the current directory, the directories specified in the PLB\_PATH environment variable are searched for the existence of both {txt} and {aam} files.

{Lnnnn} specifies the longest record length. This parameter is required when creating or indexing a null file or when indexing a variable record length or space compressed file and the first record is not the longest record in the file.

If {Lnnnn} is not specified on the command line, the record length is determined by the first record in the data file. If the input file is of fixed length records, all records in the file are assumed that length during the SUNAAMD procedure.

**SUNAAMDX – continued**

{opts} supports the following options:

<b>Option</b>	<b>Meaning</b>
D={c}	The ASCII character specified is the wildcard or don't care character for READ operations ('?' is the default).
I	Display the command line that constructed the AAM file.
I{ininame}	When specified, the utility will first look for any operational keywords in {inifile}. If the keyword is not found, the UET is searched. The filename specified by {ininame} must be a fully qualified name. If a full path is not specified, the filename must exist in the current working directory.
N	Disable the record counter display.
P{nn}{= #}'xxx'	The primary record selection criteria for partial indexing - 'xxx' may be up to eight (8) characters long.
Q	Quit without creating the output file if the input file does not exist.
R	Rebuild the index using stored command line.
S	Allow space-compressed records in the data file.
U	Treats Upper/Lower case characters equally in all READ operations.
V	Allow variable length records.
W	Write all new records at the end of the {txt} file.
Y	Do not place {txt} drive information into AAM header.
Z	Do not place {txt} drive or path information into AAM header.

**SUNAAMD**

The wildcard or don't care character defaults to a question mark but may be changed by using the D option.

P{nn}{=|#}'xxx' is the primary record select option. It allows only records meeting the specified criteria to be indexed. The criteria match, up to eight bytes, starts at column 'nn' and the match is based on the data being either equal to (=) or not equal to (#), the specified characters ('xxx'). Records meeting the criteria are indexed while the others are ignored.

Records in the data file need not be in fixed and/or non-space compressed format. If the 'S' option is given, the records may be both variable length and space compressed. If the 'S' option is given, the 'W' option is assumed.

Records written to the data file may be variable length and not be space compressed by using the 'V' option. As with the 'S' option, the 'W' option is assumed.

The 'W' option causes new records added to the file to be written at the end of file at all times. Deleted record space is not re-used and this space is not recovered until the file is reformatted.

It is possible to have all lower case letters treated as upper case during the search by specifying the 'U' option. This causes all lower case letters to be indexed as if they were upper case and treated as such during AAM I/O.

Up to 95 key specifications may be given and they may be continued to subsequent lines by terminating the line with a colon (:). Additional key specifications must then be given.

Individual key specifications must give a starting and ending record position, separated by a hyphen (-) unless a one byte key and must be separated by commas.

**SUNAAMDx – continued**

An 'X' immediately preceding an AAM key range (i.e., X1-10 or X10-20) indicates an excluded field. These fields may be used in conjunction with one or more valid AAM keys to retrieve records, but are not hashed into the AAM file. Their key position is relative to the other valid or excluded keys given in the keyspecs.

The data file is not re-written to discard any deleted record space. Only a new AAM file is created.

The order in which your fields are specified is critical during the READ operation. The first key specification is field number 1, the second is field number 2, etc.

Sub-fields are allowed. A key field may be wholly or partially contained within another field. Field 1 may be positions 1-10, while field 2 may be positions 3-5 and field 3 may be 7-14.

The information about the index file may be retrieved by specifying the index file name followed by the 'I' option. The command that was required to generate the file is displayed.

The index file may be re-indexed without knowing any of the key specs by specifying the index file name followed by the 'R' option.



## **SUNINDEX**

The SUNINDEX utility creates an ISAM key file using the specified key positions in the associated data file. SUNIDXNT is a Windows NT or Windows 95 Version of SUNINDEX. SUNINDEX uses one of the following command lines:

```
SUNINDEX [{-tnnnn}] {txt}[,{isi}[,{Lnnnn}[,{work}]]]  
- [{opts}]{keyspecs}
```

```
SUNINDEX {isi2} -[I | R]
```

If no extension is specified for {txt}, the extension TXT is assumed.

If the {txt} file does not exist, it is created if the {Lnnnn} specification is given.

If {isi} is not specified, the file name in {txt} is used with the extension ISI. If {isi} is specified without an extension, ISI is used.

If neither {txt} nor {isi} are located in the current directory, the directories specified in the PLB\_PATH environment variable are searched for the existence of both {txt} and {isi} files.

{Lnnnn} specifies the longest record length. A maximum record length of 32k bytes is supported. This parameter is required when creating or indexing a null file or when indexing a variable record length or space compressed file and the first record is not the longest record in the file.

If {Lnnnn} is not specified on the command line, the record length is determined by the first record in the data file. If the input file is of fixed length records, all records in the file are assumed that length during the SUNINDEX procedure.

**SUNINDEX – continued**

In addition to the actual key specifications, {keyspecs} supports the following options:

Option	Meaning
D	Allow duplicate keys in the ISAM key file (default).
E	Create a file containing any duplicate keys found during indexing.
F={nn}	Records are fixed length binary with NO end of record bytes.
G={nn}	Records are fixed length binary with normal end of record bytes.
N	Do not allow duplicate keys in the ISAM key file. If any duplicate keys are found, the output file is deleted. Use the 'E' option to create an error file containing duplicate key information.
NO	The index file is created ignoring duplicate keys and not adding them into the file. For PLB OPEN operations, the IFILE is tagged as a NODUP file.

**SUNINDEX – continued**

Option	Meaning
Q	Quit without creating the output file if the input file does not exist.
S	Allow space-compressed records in the data file. The 'V' and 'W' options are assumed.
U	Convert lower case characters to upper case for the ISAM key file.
V	Allow variable length records. If the 'S' option is not given, these records cannot be space compressed. The 'W' option is assumed.
W	Write all new records at the end of the file. Assumed if the 'S' or 'V' options are specified.
X	Disable record counter display.
Y	Do not place {txt} drive information into ISI header.
Z	Do not place drive or path information into ISI header.

**SUNINDEX – continued**

Records in the data file need not be in fixed and/or non-space compressed format. If the 'S' option is given, the records may be both variable length and space compressed. If the 'S' option is given, the 'W' option is assumed.

Records written to the data file may be variable length and not be space compressed. The 'V' option is used if this is the case. As with the 'S' option, the 'W' option is assumed.

The 'W' option causes new records added to the file to be written at the end of file at all times. Deleted record space is not re-used and this space is not recovered until the file is reformatted.

It is possible to have all lower case letters converted to upper case for the ISAM key file by specifying the 'U' option as the first item in {keyspecs}. This does not affect the data in {txt}, only the key information that is written to the {isi} file.

Each part of key specification must give a starting and ending record position, separated by a hyphen (-) unless a one byte key and must be separated by commas.

The sum of the key specifications cannot exceed 99 positions.

The data file is NOT re-written to discard any deleted record space. Only a new ISAM key file is created.

The information about the ISAM file may be retrieved by specifying the ISAM file name followed by the 'i' option. The command that was required to generate the file is displayed.

The ISAM file may be re-indexed without knowing any of the key specs by specifying the ISAM file name followed by the 'r' option.

## **SUNLIST**

The SUNLIST utility outputs the contents of a text file on the screen, the system printer, or to a print image disk file. The command line syntax for SUNLIST is:

```
SUNLIST {input}[, {start}[, {spool}]][-{options}]
```

If a drive specification is not given on {input}, the current path is searched followed by all search paths.

If a file type is not given, it defaults to TXT with the following exceptions:

Option	Defaults to
A	AAM
F	LST
G	LST
I	ISI

Records may be of variable length and space compressed.

**SUNLIST – continued**

The allowable parameters for {start} are as follows:

<b>Parameter</b>	<b>Indicates to start listing at ...</b>
L{nnn}	logical record number {nnn} within the text file. This option works with all types of files supported.
R{nnn}	physical sector number {nnn} within the text file. A length of 256 bytes is used for the sector length. This only works with print files or standard text files.
P{nnn}	page number {nnn} within the print format file. This option is only applicable to print files.

**SUNLIST – continued**

Allowable {options} are as follows:

Option	Meaning
A	List via AAM file. A key specification must be given and enclosed within single quotes or no records are displayed. Up to five (5) AAM keys may be given and each must be preceded by the 'A'. Under Linux, the single quotes must each be preceded by a \.
B{prog}	Chain to program {prog} at the conclusion of the list program.
C{n}	Print {n} copies. Default is one.
D	Display on terminal (default option). When displaying a file on the terminal, the F1 key pauses the display and then continues it when it has been pressed a second time. The F2 or ESCAPE key terminates the display of the file.
E{x}	Replace {x} with an ESCAPE (Hex 1B) when printing the data file. This allows for control characters to be embedded within the data file for printer control. This option is only checked if the 'F' option is specified and is valid for all output devices except the screen.
F	List a print formatted file.
G	List an ANSI print formatted file.

**SUNLIST – continued**

Option	Meaning
H{n}	Pause for {n} number of seconds after the list is finished.
I	List via ISI file. All records are listed unless a beginning key or line number is entered. If a beginning key is specified, it must immediately follow the 'I' option and must be enclosed in single quotes. Under Linux, the single quotes must each be preceded by a \.
J	Force a leading form feed before printing the file.
K	Delete the file after completion. This option is not valid with ISI and AAM files.
L	List on the system printer.
N{n}	Set the number of lines per page. This determines the number of lines written before a page advance. This option is not valid with print format files.
P	Output to a print format file.
Q	Append to an existing print format file. If the file is not found, it is created.
R	List by page in reverse order. Read the file from end of file forward (useful for laser printers).
S{n}	Indicates the starting byte location within each logical record. This then becomes the first character printed or displayed.
T{n}	Tab to column {n} before printing each line.
V	Suppress final form feed.
W	Wrap option. List the entire record, even if it exceeds the output medium length.
X	List without line numbers.
Z{n}	Execute a Sunlist command upon completion of the list.



## **SUNSCAN**

The SUNSCAN utility displays any part of a file using the arrow keys and page controls. SUNSCAN may also search for strings in the file and display a screen of information following the search string.

```
SUNSCAN {filename} [-{opt}]
```

The Left, Right, Up and Down arrow keys move the display in the appropriate direction. PgUp and PgDn display the next or previous screen. Home and End display either the top or bottom line while still maintaining the current column position.

The ESCAPE terminates the program.

{opt} supports the following option:

Option	Meaning
W{nn}	specifies the line width. If not specified, the line width is 250 bytes. It may be changed to any value between one and 65535.
S{"string"}	specifies an initial search string.

The 'S' key initiates a search. A window is opened allowing entry of the search string. A status line is displayed on the first line of the screen giving information concerning the progress of the search and the location in the file where the information was found when the search succeeds.

## **SUNSORT**

The SUNSORT utility sequences a data file in a particular order. The command line syntax for SUNSORT is as follows:

```
SUNSORT [-L{log}] [{-tnnnn}]{input}[, {output}  
[, {tmppath}[, {sortseq}]]] -[{opts}, ]{keys}
```

If a path specification is not given on {input}, the current path is searched followed by all search paths.

When no file extension is given for {input} or {output}, the default text extension as specified in the screen definition file is used.

If neither {input} nor {output} are located in the current directory, the directories specified in the PLB\_PATH environment variable are searched for the existence of both {input} and {output} files.

If {output} is not specified, the file is sorted to a temporary file and when the sort is finished the temporary file is renamed to the original file.

If a path specification is not given on {output}, the file is placed in the current path.

If the log file option is specified, all program messages are written to {log}.

**SUNSORT – continued**

{opts} refers to special options that affect the output of the sort.

Option	Meaning
B	Treat any space compression characters as binary characters and do not expand as a compression count.
F={nn}	Records are fixed length binary with NO end of record bytes.
G={nn}	Records are fixed length binary with normal end of record bytes.
K	Key tag file output. The output of a key tag sort is a file with record pointers to where the sorted records are located in the input file. The file is in a binary format with each pointer taking up 4 bytes. There are no carriage returns or line feeds in the file. The file may be read with PL/B by reading an INTEGER 4 field followed by a semicolon to keep the record position at the current location.
S{nn}{= #}{c}	Selective output. If specified, only the records that match the specification are included in the output file. Since only a single character may be specified for this option, multiple S options select on multiple record positions. The {nn} specifies the record starting position, the {= #} specifies either an equal or not equal condition test, and the {c} is the character to compare.

**SUNSORT – continued**

{keys} refers to the actual key information used in the sequencing of the file. Any key specification may be preceded by one of the following options. The options affect every key specification that follow until an alternate option is given.

Letter	Meaning
A	(default) All data within this keyspec range is sorted in ascending order.
D	(optional) All data within this keyspec range is sorted in descending order.
L	(default) All lower case data within this keyspec range is treated as lower case.
U	(optional) All lower case data within this keyspec range is treated as upper case.
N	The sort key is sorted as a signed numeric. When the minus sign character identifies a negative value, the sort output is in reverse order for negative numeric keys.

## **TXTMATCH**

The TXTMATCH utility compares two text files for any inconsistencies between them. When a discrepancy is encountered, the comparison stops for your review. Each file is displayed separately on one-half of the screen display.

```
TXTMATCH {infile1},{infile2}[, {outfile}] -[{options}]
```

If the second file specification contains only a path, the input file name is assumed.

The program options control the handling of the log file:

Option	Meaning
A	Create the log file
Q	Append to the log file

When two files have been specified, the display of the files may be advanced individually or concurrently by use of the function keys.

Key	Meaning
F1	Read the first file and advance the display accordingly.
F2	Read the second file and advance the display accordingly.
F3	Read both files and advance the display accordingly.

**TXTMATCH – continued**

When three files have been specified, the display of the files and the output to the third file may be advanced and controlled by the use of the function keys. When the corresponding records in the first two files match, the record is automatically output to the third file. Only when the files do not match do the following function keys become active.

Key	Meaning
F1	Skip record in file 1.
F2	Skip record in file 2.
F3	Skip records in both file1 and file2.
F5	Write current record in file 1 to file 3 and read another record from file 1.
F6	Write current record in file 2 to file 3 and read another record from file 2.
F7	Write current record in file 1 to file 3 and read another record from both files.
F8	Write current record in file 2 to file 3 and read another record from both files.