GuptaPro

Hot ideas for Gupta® Developers

Static Control for SQLWindows

R. J. David Burke

A common question on Gupta's CompuServe forum is, "How do I change the title of a background text control at runtime?" The author offers techniques for manipulating background text and other static controls during the execution of a SQLWindows program. The article also reveals new features in SQLWindows 5.0.2 that simplify static control manipulation.

You implement most controls (data fields, push buttons, list boxes, and so on) in SQLWindows as actual Windows objects. As an object in the Windows environment, a control can be accessed by its window handle. Through its window handle you control the appearance and behavior of a control through the functions and messages provided by SQLWindows (and Windows itself).

Each object in the Windows environment uses some of Windows' System Resources, which are limited. A large application with many windows and controls can easily bring Windows to its knees by reducing the available resources to a critical level.

Gupta addressed this issue in version 4.0 of SQLWindows by giving developers an alternative to creating certain types of controls as Windows objects. The alternative is to "paint" these types of controls on top-level windows, like a bitmap, instead of creating these controls as Windows objects. This alternative is available for controls known as *statics*.

What are statics?

In SQLWindows, statics are background text, lines, and frames. These are controls that typically don't change their appearance at runtime, and they have no behavior or functionality. They're sized and positioned at design time and continue to stay with their design-time settings at runtime.

A common misconception is that group box controls are also statics. These are, indeed, passive controls (they don't process messages), but they aren't statics, mostly for historical reasons. Group boxes are actually implemented in Windows as a type of push button.

To implement statics, SQLWindows 4.0 introduced a new global system

Continues on page 3

Premier Issue

Number 1

1 Static Control for SQL Windows *R.J. David Burke*

2 Whole Lotta Shakin' Goin' On *Mark Hunter*

7 Pick Up the Tabs *Mark Hunter*

9 Unravel the Black Art of Scoping in Windows *Jolyon Smith*

12 Get Connected with Hierarchies *Michael Cunningham*

Whole Lotta Shakin' Goin' On

Mark Hunter

When we at Pro Publishing began planning this newsletter, we knew that Gupta Corp.'s customers were in need of more information from and about the company. As of January 2, that need has intensified. Virtually everything about Gupta is changing in a massive restructuring announced that day. Umang Gupta has announced plans for retirement as chairman in May of 1996, and Sam Inman has been named CEO. The company has laid off 17 percent of its employees (engineering is unaffected, according to our sources; the cuts came from other departments). The company's name will change from Gupta Corporation to Centura Software Corporation. Matt Miller becomes V.P. of Marketing (the position had been empty for many months). Press releases are notorious for clichés, but this one strikes amens from many developers with the phrase, " . . . transitioning the company from a technology vision-driven enterprise to one that is more market-driven."

Money talks, and sometimes money screams. This restructuring is likely the result of several consecutive quarters of losses in spite of generally increasing revenue. The company's stock price has dropped dramatically, and the call for short-term profit–regardless of the long-term strategy–is impossible to ignore any longer. How will that profit be attained? Mostly by slashing costs, in the form of payroll and other expenses.

When it comes to revenues, Centura is the great hope of the company. Gone is the promise made in August (at the Windows 95 rollout) that there would be a 32-bit SQLWindows in 90 days. Gupta subsequently decided that customers would prefer an earlier release of the new Centura tools and could wait until then for 32-bit tools. (They also saved money by not doing a separate, release of 32-bit SQLWindows.) The company assures us that Centura is a new product, deserving of a separate brand name—and a new pricing structure. Existing Gupta customers will get a discount; customers with current LSS contracts will get an additional discount; but the numbers have yet to be finalized.

Centura, we're told, is full of must-have features beyond those currently found in SQLWindows: a better compiler, multi-platform capabilities, Internet integration, support for three-tier architecture, a vastly improved TeamWindows, and others. As we go to press we're under non-disclosure agreements, but upcoming issues of *Gupta Pro* will have in-depth coverage of Centura.

Gupta staff are sticking to the company line, which says that Centura is an next-generation product. Is it really a generation ahead of SQLWindows 5.0.2? Probably not. But I, for one, am willing to buy into the company line. SQLWindows has always been an excellent product, and Gupta's accompanying marketing efforts have always been lackluster. An entirely new marketing effort is supposedly being planned for Centura, and it would certainly be welcome. Gupta's recent visibility has been quite subdued; one hopes that it's marshaling its

Continues on page 15

Editor Mark Hunter, **Publisher** Dian Schaffhauser, **Vice President of Slobber** Mocha

Gupta Pro is published monthly (12 times per year) by Pro Publishing, PO Box 18288, Seattle, WA 98118–0288.

POSTMASTER: Send address changes to *Gupta Pro*, PO Box 18288, Seattle, WA 98118–0288.

Copyright © 1996 by Pro Publishing. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever (except in the case of brief quotations embodied in critical articles and reviews) without the prior written consent of Pro Publishing. Printed in the United States of America.

Gupta Pro is a trademark of Pro Publishing. Other brand and product names are trademarks or registered trademarks of their respective holders.

This publication is intended as a general guide. It covers a highly technical and complex subject and should not be used for making decisions concerning specific products or applications. This publication is sold as is, without warranty of any kind, either express or implied, respecting the contents of this publication, including but not limited to implied warranties for the publication, performance, quality, merchantability, or fitness for any particular purpose. Pro Publishing, shall not be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this publication. Articles published in *Gupta Pro* reflect the views of their authors; they may or may not reflect the view of Pro Publishing.

Subscription information: To order, call Pro Publishing at 206-722-0406. Cost of domestic subscriptions: 12 issues, $129; 24 issues, $199. Canada: 12 issues, $144; 24 issues, $229. Outside North America: 12 issues, $159; 24 issues, $259. Individual issues cost $12 ($15 outside the U.S.). All funds must be in U.S. currency.

Gupta technical support: Call Gupta Corp. at 415-321-4484/

If you have other questions, ideas, dog food, or would just love to chat about what you're doing with Gupta, contact us via one of the means at right.

We're Standing By!

Editorial Headquarters
818-249-1364

E-mail
71460.3142@compuserve.com

Fax
206-760-9026

Mail
Pro Publishing
PO Box 18288
Seattle, WA 98118-0288

Source Code
GO GUPTA, Library 10

Static Control . . .

Continued from page 1

variable named bStaticsAsWindows. This is a Boolean variable and can have a value of either TRUE or FALSE. The initial value of this variable is FALSE.

When bStaticsAsWindows is TRUE, statics are created as Windows objects at runtime (as in SQLWindows version 3 and prior). When bStaticsAsWindows is FALSE, statics are painted on SQLWindows form windows and dialog boxes. Each time SQLWindows has to create a top-level window at runtime, the bStaticsAsWindows variable is examined; its value determines whether the statics on the newly-created window will be Windows objects or simply painted on the window.

Using the bStaticsAsWindows variable

If your application doesn't need to manipulate statics, then you really don't need to do anything further. The bStaticsAsWindows variable is initialized to FALSE by SQLWindows at the start of a program's execution, so the default of painting statics will apply.

If you need to manipulate statics during the course of execution, you can set bStaticsAsWindows to TRUE. Typically, this is done in the SAM_AppStartup message processing in Application Actions. For example:

```
◆ Global Declarations
  .
  .
  .
  ◆ Application Actions
    ◆ On SAM_AppStartup
      ◇ Set bStaticsAsWindows = TRUE
```

Although atypical, you can change the value of bStaticsAsWindows throughout your application, creating some top-level windows with statics as Windows objects and other top-level windows with statics painted. For example:

```
◆ Global Declarations
  .
  .
  .
  ◆ Application Actions
    ◆ On SAM_AppStartup
      ◇ ! bStaticsAsWindows is FALSE
            by default
      ◇ ! the following window will have
            its statics painted
      ◇ Call SalCreateWindow( frm1, hWndNULL )
      ◇ Set bStaticsAsWindows = TRUE
      ◇ ! now statics will be created as
            Windows Objects
      ◇ Call SalCreateWindow( frm2, hWndNULL )
```

At the time a top-level window is created, the current value of bStaticsAsWindows determines whether statics will be created as Windows objects.

The advantages of painting statics

If you don't need to manipulate statics, there are two advantages to leaving bStaticsAsWindows at its default setting of FALSE. First, you decrease your application's use of Windows' System Resources. This leaves more resources for creating other Windows objects within your application and provides resources for other Windows applications concurrently running with your application.

Second, you'll obtain increased performance. SQLWindows can create and display top-level windows faster when statics are painted.

The SQLWindows application STATIC1.APP (provided through the online sources listed on Page 2) demonstrates both of these advantages. When you start this application, you're asked if you want to set bStaticsAsWindows to TRUE in the startup dialog box. Run this application twice, first with bStaticsAsWindows set to TRUE and then set to FALSE. Depending on your hardware, you should notice a distinct speed difference in creating the top-level form windows. During the execution of the application, switch back to SQLWindows and choose View | Statistics... from the menu. You should see a drastic difference in the amount of User Resources (part of System Resources) available. Figures 1 and 2 (see page 4) show the SQLWindows Statistics dialog when bStaticsAsWindows is left at its default FALSE setting and then set to TRUE.

The disadvantage of painting statics

The disadvantage of painting statics is that you can't directly manipulate static controls at runtime. Since statics are painted, they lack a window handle. This prohibits you from moving them, resizing them, hiding them, and so on. That is, until SQLWindows 5.0.2 was released. In version 5.0.2 Gupta added several SAL functions that enable you to manipulate statics directly (which I'll discuss shortly).

You won't have to be concerned with manipulating statics at runtime in basic SQLWindows applications; but it will come up in more sophisticated and advanced applications, when you want to resize a frame or reposition a line. You may also want to hide/show statics at different states of the application. And the most common manipulation of statics at runtime is to change the title of a background text control. To do these kinds of manipulations with versions of SQLWindows up to and including SQLWindows 5.0.0, you'll have to live with setting bStaticsAsWindows to TRUE so that statics are created as Windows objects and have a window handle.

Gupta provides the ability to do certain manipulations indirectly on background text controls when used as a label for other SQLWindows controls. I'll also cover this later.

Manipulating statics at runtime

As long as statics are created as Windows objects (bStaticsAsWindow was TRUE when the window was created), you can manipulate statics at runtime by

locating their window handles. This is accomplished by using the SalGetFirstChild and SalGetNextChild functions. For example:

```
◇ Set hWnd =
    SalGetFirstChild( hWndForm, TYPE_Line )
```

This example searches the specified top-level window (hWndForm) for the first line control and returns its window handle (or hWndNULL if no line controls are found). The window handle is then assigned to the variable hWnd. Once you have the window handle, you can manipulate the line. For example, you would hide the line with the following function call:

```
◇ Call SalHideWindow( hWnd )
```

If a top-level window has more than one of a given type of control, you can use SalGetNextChild to locate the subsequent controls. To hide all the lines and frames of a top-level window, you could use the following code:

```
◇ Set hWnd = SalGetFirstChild( hWndForm,
             TYPE_Line | TYPE_Frame )
◆ While hWnd != hWndNULL
        ◇ Call SalHideWindow( hWnd )
        ◇ Set hWnd = SalGetNextChild( hWnd,
             TYPE_Line | TYPE_Frame )
```

Notice that SalGetFirstChild uses a top-level window as its first parameter while SalGetNextChild uses a child control as its first parameter.

Once you have the window handle of a control, you can use the functions in Table 1 to manipulate the control:

Table 1. Sal functions to manipulate controls.

Function name	Use
SalBringWindowToTop	Bring a control to the top of the z-order
SalCenterWindow	Center a control relative to its parent
SalDisableWindow	Disables a window for user input
SalEnableWindow	Enables a window for user input
SalGetWindowLoc	Get the location coordinates of a control
SalSetWindowLoc	Set the location coordinates of a control
SalMoveWindow	Move a control relative to its current position
SalGetWindowSize	Get the size of a control
SalSetWindowSize	Set the size of a control
SalHideWindow	Hide a control
SalShowWindow	Show a control
SalIsWindowVisible	Determine if a control is visible
SalColorGet	Get the color of a control
SalColorSet	Set the color of a control
SalGetWindowText	Get the text or title of a control*
SalSetWindowText	Set the text or title of a control*
SalFontGet	Get the font characteristics of a control*
SalFontSet	Set the font characteristics of a control*

* Not applicable to lines or frames

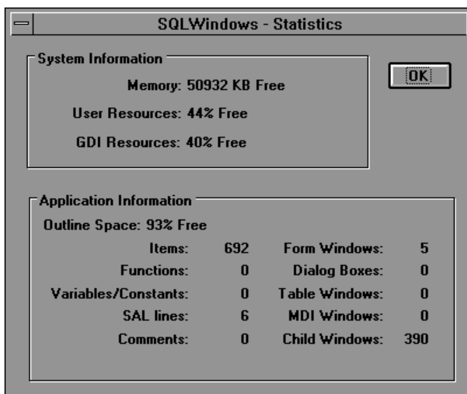

Figure 1. The System Resources available when running STATIC1.APP with bStaticsAsWindows left to its default FALSE setting . . .

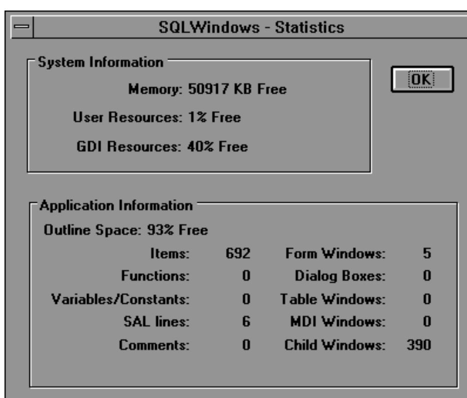

Figure 2. . . . and when set to TRUE.

The problem with this approach is that you don't know which control you have the window handle of. Background text, lines, frames, and group boxes don't have template names, so you need to know something about the arrangement of these controls to know which control you're dealing with. For background text you can check the text (SalGetWindowText). For lines and frames, you can check the location and size information. But these techniques require that the application know about the physical layout of its controls. However, there isn't really any other solution.

For background text, one alternative is to use a data field. You can define a data field with no border and change its background color to match the underlying window. This can be a fairly effective technique, because you can then refer to a data field by its name and manipulate it without searching for its window handle. The caveat of this technique is that you can't set a mnemonic underline in the text of a data field.

Background text as labels

One of the features of the background text control is that

it can be used as a label for another control, such as a data field, a list box, or just about any other SQLWindows Control. To take advantage of this feature, simply position the background text control to immediately precede the complementary control in the application outline, for example:

```
◇ Background Text: &Data Field
◆ Data Field: df1
    ◇ Message Actions
```

Here, the background text with the title "&Data Field" becomes a label for the data field df1 because it immediately precedes df1 in the outline.

The physical position of the background text on the design window in relation to the complementary control is irrelevant to SQLWindows. It's the application outline order that determines what control the background text becomes a label for.

Labels provide a mnemonic keyboard accelerator

Notice the ampersand (&) in the preceding example that precedes the "D" in "data field." An ampersand in a background text control does two things:

1. The character following the ampersand is underlined in the actual background text on the design window.

2. SQLWindows automatically sets up a keyboard accelerator for the complementary control allowing the user to use Alt key in conjunction with the alphabetic key. In the example, if you pressed Alt-D, the focus would change to the data field df1 as the complementary control for the background text.

Using an ampersand to set up a keyboard accelerator is a standard Windows technique. It also works for menu items, push buttons, and group boxes.

When you use a background text control as a label, SQLWindows provides additional programming support to manipulate the label. Table 2 shows functions you can use to manipulate a control and its label in one stroke.

Table 2. Sal functions to manipulate controls and their labels.

Function name	Use
SalDisableWindowAndLabel	Disables the control and dims its label
SalEnableWindowAndLabel	Enables the control and normalizes its label
SalHideWindowAndLabel	Hides the control and its label
SalShowWindowAndLabel	Shows the control and its label
SalGetWindowLabelText	Retrieves the text of the label for a control
SalSetWindowLabelText	Sets the text of the label for a control

There are advantages to using these functions. When bStaticsAsWindows is FALSE and background text is painted, these functions are the only way you can manipulate the label. When bStaticsAsWindows is TRUE, these functions manipulate the label along with the complementary control, eliminating the need to use SalGetFirstChild/SalGetNextChild to find the window handle of the background text.

The SalGetWindowLabelText and SalSetWindowLabelText functions let you manipulate the text of the label at runtime. You can even change the keyboard accelerator by using an ampersand in your text with SalSetWindowLabelString.

The SQLWindows application STATIC2.APP (see Page 2 for details on obtaining it) demonstrates the use of labels and the functions that manipulate the labels. It also reveals some interesting anomalies when using labels. For example, when bStaticsAsWindows is set to TRUE, picture controls fail to pick up their label from the outline. Why? Because Picture controls are always created last, regardless of their position in the application outline. If bStaticsAsWindows is set to TRUE, the background text that precedes the picture control is created before the picture is aware of it and the picture ends up without a label. With picture controls you also need to set its editable customizer attribute to Yes so it can receive the focus when the complementary accelerator is used. Figure 3 shows the STATIC2.APP application.

Another interesting note from the application is that in order for custom controls to work with keyboard accelerators, the tab stop setting in the custom control's customizer must be set to either tab or group. Leaving it at the default setting of None will exclude the custom control from receiving keyboard accelerators.

Also note that radio buttons and check box controls show anomalous behavior when bStaticsAsWindows is set to FALSE. Keyboard accelerators for radio buttons are ignored and check boxes get checked in this situation. Fortunately, you can put the accelerators right in these controls to get around this problem. Actually, the STATIC2.APP application is atypical in this regard, since you don't usually use labels with these types of controls.

For the curious, I've included an undocumented SQLWindows function called SalGetWindowLabel, which returns the window handle of the background text label for a specified control, for example:

```
◇ Set hWnd = SalGetWindowLabel( df1 )
```

Upon return from this function, the variable hWnd will contain the window handle of the background text control that is the label for df1. Of course, this only works when bStaticsAsWindows is set to TRUE when the window was created. If bStaticsAsWindows was FALSE, the hWnd variable would have the value of hWndNULL.

New functions for static control

The Sal functions I discussed in the previous section make manipulating background text possible when bStaticsAsWindows is set to FALSE. But what about manipulating other statics—specifically, lines and frames—when they're painted? Previous to SQLWindows 5.0.2 there was no provision for this kind of manipulation. SQLWindows 5.0.2 includes a handful of new functions that allow for manipulating statics at runtime, as listed in Table 3.

Table 3. Sal functions to manipulate controls and their labels.

Function name	Use
SalStaticFirst	Returns a "handle" to the first static on a window
SalStaticNext	Returns a "handle" to the next static on a window
SalStaticGetLoc	Gets the location coordinates of a static
SalStaticSetLoc	Sets the location coordinates of a static
SalStaticGetSize	Gets the size of a static
SalStaticSetSize	Sets the size of a static
SalStaticHide	Hides a static
SalStaticShow	Shows a static
SalStaticIsVisible	Determines whether a static is hidden
SalStaticGetLabel	Gets the static "handle" of the label for a specified control

The file STATIC3.APP (included with this month's online source code) demonstrates using some of these functions. Figure 4 shows the application at runtime.

Internally, SQLWindows maintains a list of statics for each top-level window. This list provides SQLWindows with the information it needs to repaint statics when a window needs to be redrawn. The SalStaticFirst function takes a top-level window handle as a parameter and returns a number that's used like a handle. This handle is the first static in the SQLWindows internal list of statics for the specified top level window; for example:

```
◇ Set hStatic = SalStaticFirst( hWndForm )
```

The variable hStatic is a numeric variable, but I've prefixed it with "h" instead of the usual "n" to indicate that it's a handle.

Once you have the handle of a static, you can manipulate the static with the other SalStatic* functions described later. But before you do that, you should also know about the SalStaticNext function. SalStaticNext returns the handle of the next static in the internal list for a specified top-level window:

```
◇ Set hStatic =
     SalStaticNext( hWndForm, hStatic )
```

In this line of code, the SalStaticNext function receives a window handle and a static handle. It returns the next static handle, which is assigned to the hStatic variable. If the return value is 0, there are no more statics in the internal list for the specified top-level window. The SalStaticFirst and SalStaticNext functions work very much along the lines of SalGetFirstChild and SalGetNextChild.

Now that you know how to navigate through static handles, you can use the other SalStatic* functions to manipulate the statics.

SalStaticGetLoc and SalStaticSetLoc can be used to programmatically move a static at runtime. The syntax for these functions is:

```
◇ Set bOk = SalStaticGetLoc( hWndForm,
              hStatic, nXCoord, nYCoord )
◇ Set bOk = SalStaticSetLoc( hWndForm,
              hStatic, nXCoord, nYCoord )
```

The nXCoord and nYCoord variables are pixels relative to the upper left corner of the top-level window.

You can use SalStaticGetSize and SalStaticSetSize to size a static at runtime programmatically. The syntax for these functions is:

```
◇ Set bOk = SalStaticGetSize( hWndForm,
              hStatic, nWidth, nHeight )
◇ Set bOk = SalStaticSetSize( hWndForm,
              hStatic, nWidth, nHeight )
```

The nWidth and nHeight variables are pixels relative to the location of the static.

SalStaticHide, SalStaticShow, and SalStaticIsVisible access and manipulate the visibility of the static:

```
◇ Set bOk = SalStaticHide( hWndForm, hStatic )
◇ Set bOk = SalStaticShow( hWndForm, hStatic )
◇ Set bVisible = SalStaticIsVisible
                    ( hWndForm, hStatic )
```

Finally, SalStaticGetLabel returns the static handle for the label of the specified control. The following example finds the static handle of the label for the data field df1:

Continues on page 16

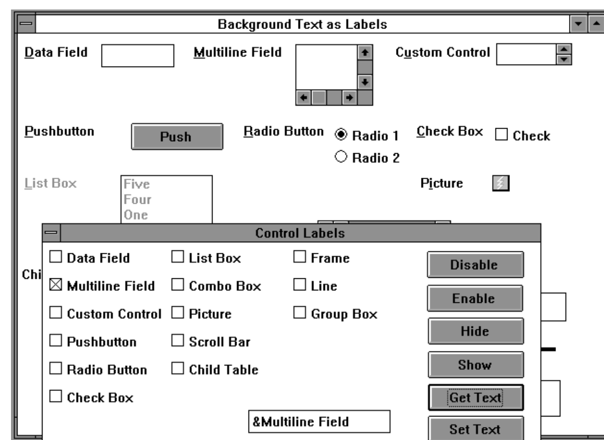

Figure 3. The STATIC2.APP application at runtime. Use the modal dialog box to manipulate the labels in the main window.

Pick Up the Tabs

Mark Hunter, Editor

ReportWindows has been the source of frustration for developers who love its tight integration with SQLWindows and despair at its limited feature list. It has probably inspired more kludges than any other Gupta tool. Well, here comes another one! A minor feature in ReportWindows provides a surprising degree of flexibility and sophistication for your reports.

If you've looked at other report writers, you probably realize that ReportWindows has some big gaps in its functionality. Yet no other report writer works nearly as well in cooperation with SQLWindows. One of the most pressing needs has been the ability to handle multiple, different sets of detail data within the same report. Many report writers allow multiple sets of data to be presented easily, in a variety of physical layouts. ReportWindows makes it easy to use one set of detail data, and very difficult to use more than one. Gupta Corporation has promised to make the handling of multiple sets of data much simpler, but in the meantime we must deal with ReportWindows in its current form. This has lead to a variety of workarounds (some would say kludges) to present multiple sets of data to the user without hinting at the struggles that we went through.

Most of these workarounds involve rather large and complex class structures, with related line formulas in the ReportWindows QRP files. They provide control over the manipulation of each item in each set of data. Such applications take some time to learn and will be the focus of another article, not this one. Right now, I'm going to share a quick technique that, despite its drawbacks, creates good-looking reports in very little time. Thanks go to Mark Culverhouse of Gupta Corporation for the inspiration for this article.

Better living through tab stops

Let's cut to the chase. Have a look at the report in Figure 1. This is a simple report produced from the sample database tables in the SPA application that ships with SQLWindows. A couple of things should be immediately apparent:

- Multiple blocks of detail data are being presented in a columnar format.
- The blocks are side by side. That's unusual even in the more complicated workarounds.

The short explanation is this: Each "block" of data is actually a single field in the QRP file. All the columnar formatting and multiple lines in each block are accomplished by inserting tabs and carriage returns into this field's input item. This happens inside the SQLWindows application before the app passes the value of the item to ReportWindows. The QRP file, in turn, has special tab stops specified for the field. These tab stops work in conjunction with the tabs embedded in the field's value.

How did we do it?

The sample application, TABSPA.APP, in Listing 1, fetches all names from table GUEST into a hidden table window, then moves through that table window one row at a time in response to SAM_ReportFetchNext messages. Before returning from this message, the app must get and format the two complicated strings that will be passed the ReportWindows input items TREATMENTS and WEIGHTS.

Okay, fine. Listing 1 shows a reasonably small piece of code. But how do GetWeighIns() and GetTreatments() create that string with all that complex formatting? Look at Listing 2.

The eventual look of the "Treatments" block in Figure

TABSPA.QRP - Demonstration of Tabbed Fields

Guest Name	Treatments...	Start	Stop	Price	Weigh-ins...	
BETTE MIDRIFF						
	WEIGHTS	10/11/92	10/11/92	100	10/11/92	161
	JOGGING	10/11/92	10/11/92	0	10/12/92	159
	AEROBICS	10/11/92	10/11/92	40	10/14/92	150
					10/16/92	148
BIANCA JOGGER						
	ACUPUNCTURE	10/6/92	10/6/92	150	10/6/92	113
	RAQUETBALL	10/6/92	10/6/92	40	10/7/92	109
	KARATE	10/6/92	10/6/92	150		
BOZO						
	TENNIS	10/21/92	10/21/92	150	10/21/92	280
	WEIGHTS	10/21/92	10/21/92	100		
	BOWLING	10/21/92	10/21/92	10		
CLINT WESTWOOD						

Figure 1. A simple report produced from the sample database tables in the SPA application.

1 has four columns. That's why we fetch four columns from database table GUEST_TREATMENT within this function, then format them all as strings, then separate them with tabs and carriage returns.

The GetWeighIns() function works in a very similar manner, fetching two database columns from table GUEST_WT. Note that the functions expect a parameter, strName, which tells them which rows from the database are needed for this particular row in the report. They also expect parameter hSql, which they use in their queries. This is just a personal preference on my part; I stashed the guest names in a table window so that the cursor used to get guest names could be reused by the functions. If I had used two cursors instead, I could have fetched guest names with one, avoiding the use of a table window, while using the second cursor in GetTreatments() and GetWeighIns().

Listing 1. TABSPA.APP

```
[sd] On SAM_ReportFetchInit
    [hd] ! stick the guests into a table window for later processing.
    [hd] Call SalTblPopulate(tblGuests,hSql,'select name from guest',TBL_FillAll)
    [hd] Set nContext=-1
[sd] On SAM_ReportFetchNext
    [hd] ! remember that table window? Process guests one by one.
    [hd] Set nContext=nContext+1
    [sd] If SalTblSetContext(tblGuests,nContext)
        [hd] ! call a function that returns all the treatments in a single,
                complex string.
        [hd] Call GetTreatments(hSql,tblGuests.colName,strTreatments)
        [hd] ! do the same with weigh-ins.
        [hd] Call GetWeighIns(hSql,tblGuests.colName,strWeighIns)
        [hd] Return TRUE
    [sd] Else
        [hd] Return FALSE
```

Listing 2. GetTreatments

```
[sd]Function: GetTreatments
    [hd]Description:
    [hd] Returns
    [sd] Parameters
        [hd] Sql Handle: hSql
        [hd] String: strName
        [hd] Receive Long String: strTreatments
    [sd] Static Variables
    [sd] Local variables
        [hd] Number: nPrice
        [hd] Date/Time: dtStart
        [hd] Date/Time: dtEnd
        [hd] String: strTreatment
        [hd] Number: nReturn
    [sd] Actions
        [hd] ! strTreatments will contain four "columns" separated by tabs.
                Go fetch the four values from the database.
        [hd] Call SqlPrepareAndExecute(hSql,'select treatment,start_date,end_date,price
                from guest_treatment where name=:strName
                into :strTreatment,:dtStart,:dtEnd,:nPrice')
        [hd] Set strTreatments=''
        [sd]While SqlFetchNext(hSql,nReturn)
            [sd]If strTreatments>''
                [hd] ! We only append a carriage return between lines, not on the last line.
                [hd] Set strTreatments=strTreatments || CRLF
            [hd] ! strTreatments can't contain other data types - everything must be formatted
                    here in the app rather than in the report's QRP file.
            [hd] Set strTreatments=strTreatments || strTreatment || TAB ||
                    SalFmtFormatDateTime(dtStart,'M/d/yy') || TAB ||
                    SalFmtFormatDateTime(dtStart,'M/d/yy') || TAB ||
                    SalNumberToStrX(nPrice,0)
```

The dark side

Hey, this stuff's great! What are the disadvantages? Well, there are several, and they can all be traced to the trick that got us here in the first place: what looks like a detail block on the report is really just one big, cleverly formatted string. If the four columns in the "Treatments" block were really four separate fields, there are several ReportWindows features that could be used on them:

- We could assign different fonts and colors to the four fields.
- We could use their values in report formulas.
- We could apply formulas against them to decide whether to print them or not.
- We could use any data type, not just strings, and format them as we pleased.
- We could use field wrapping on especially long values.

None of these features can be used on the individual "columns" in the blocks of our sample report.

What to do?

We've learned that tabbed fields can make a large cosmetic splash, but can't equal the value of truly distinct ReportWindows input items. Some reports can take advantage of benefits and live with the limitations. In other cases, we'll still need to use those workarounds until ReportWindows' next release. **P**

Mark Hunter, Editor of *Gupta Developer*, also runs Hunter Software, Inc., a client/server consulting firm in Southern California. 818-249-1364, CompuServe 71460,3142.

Unravel the Black Art of Scoping in SQLWindows

Jolyon Smith

Scope might be defined by a SQLWindows developer this way: During the activity of an application, as roles are played out, every opportunity is taken to illustrate the tendency to aimlessly drift from the programmer's original intention. Yet, with a little discipline and knowledge of the rules, you can master scoping fundamentals.

The nature of SQLWindows makes scope more of a problem than with other development tools—variables can be declared in so many locations. In the good old days a variable was either local or global, end of story. Compare that simple distinction with the possible locations for SQLWindows variable declarations:

- Global
- Function local (includes parameters)
- Function static
- Window (includes parameters)
- Class
- Class instance

If you went further to identify the different combinations of "contained" windows (in other words, child table windows or form windows in an MDI frame), and to distinguish between "receive" parameters and regular parameters, the list would surely treble in size again!

So how do you make things easier for yourself? Well, two things can be done straight-away. First, set "Fully qualify external references?" to "Yes." Second, set "Enable runtime checks of external references?" to "Yes."

How does this help? The first setting ensures that all external references made by the program are verified by the compiler. The second enables additional checks at runtime. You'll notice that if you attempt to build an executable file from your application, you're now warned about this setting. (More on this later.)

External references

So just what is an external reference? First, it has *nothing* to do with external functions! External references in this context refer to the situation where one object makes reference to another object that isn't *directly* in scope. Still not clear? Perhaps this will help:

```
◆ Form Window: frmStart
  ◆ Contents
    ◇Pushbutton: pbCreateDlg
  ◆ Message Actions
    ◆ On SAM_Click
      ◆ Call SalModalDialog(
        dlgCustomer, hWndForm )
  ◆ Window Variables
    ◇ String: sCustID

◆ Dialog Box: dlgCustomer
  ◆ Contents
    ◇ Data Field: dfCustomerID
  ◆ Message Actions
    ◇ On SAM_Create
      ◇ Set dfCustomerID = sCustID
```

This somewhat contrived example serves to illustrate the point. dlgCustomer makes a reference to an item (sCustID) outside its own section of the outline. It refers to an object external to itself, ergo, it's an external reference. Make frmStart an "Auto Create" window to test the runtime of the application.

Try compiling this example with the "Fully qualified external references?" as "Yes." Your application should fail to compile. However, change this setting to "No" and your application will compile *and run* successfully! When the compiler encounters what appears to be an invalid reference (sCustID), it checks to see if it is *at all possible* for the reference to be correct at runtime. If the *possibility* exists, then internally some shoulders shrug and your application is left to sort it out. In my example, the compiler determines that an sCustID variable may exist (since frmStart could exist) and leaves you to it. In this case everything works just dandy—dlgCustomer can't exist with an instance of frmStart with which to invoke it. But if this dependency changes later, the application may compile successfully, but blow up at runtime.

Unqualified references

Now I want you to introduce a global variable, also a string, also called sCustID, and compile and run your application again. In this case, it doesn't matter how you set the "Fully qualified external references?" option. Whichever setting is in use, the reference will always be to the global variable version of sCustID—and you never get told this either during compilation or at runtime. Global

variables *always* take precedence over unqualified external references.

This gives a clue as to what is happening at runtime when an unqualified external reference is encountered: The runtime environment first looks for a global variable with which to resolve the reference. If no global variable exists, the environment then looks at all the *currently open* top-level window objects, looking for a variable with which to resolve the reference. Only if that fails will an error occur. You've seen it: "Unrecoverable error: ... ". Window parameters are ignored in this hunt to resolve the reference! Only variables count.

This also hints at how the *compiler* does its checking: If a single variable exists that *may* serve to resolve the unqualified reference at runtime, then the compiler will let it pass. Only if there's no such variable *anywhere* (globally or in a top-level window) or if there's more than one candidate (in other words, not globally but in more than one top-level window) will a compile time error occur asking you to provide a qualification for the reference.

This can be shown by creating another form window as follows (also "Auto Create"):

```
◆ Form Window: frmDummy
    ◆ Window Variables
        ◇ String: sCustID
```

Attempt to compile your application now, and you'll get an error. The compiler is no longer happy to let you decide at runtime which sCustID to use. If your app *did* compile and run successfully, you probably still have a global variable declared; remember, this takes precedence.

You need qualifications in this world!

How do you avoid these complications? Never rely on the runtime environment to resolve your references. "Fully qualify external references?" should *always* be set to "Yes."

If you don't do this, your application may work today; it may even work next week. But you'll be extremely lucky if it still works in a few months, when maintenance programmers have had a go at it.

When you wrote your application, you had intimate knowledge of all your variables' declarations and how they would interact and resolve at runtime. Anybody else has to rely on your excellent documentation and/or guess work. If you discipline yourself to *always* qualify your references, then two things happen: 1) Documentation becomes implicit in the code (that's no excuse for not documenting other aspects of your code, though!); and 2) as a consequence, maintenance work becomes easier and more reliable.

And we all know that the best way to discipline yourself is to get somebody else to hold the whip!—in this case, the SQLWindows compiler. This will, of course, introduce its own problems. You now have to qualify all your external references, or use global variables all the time (bad idea).

Qualification of SQLWindows variables and objects is a whole topic of its own and isn't covered here. Suffice it to say that the SQLWindows on-line help system does a more than adequate job, and if you're still stuck, then try the *Technical Reference* documentation.

Reviewing the options

I said that if "Runtime checks of external references" were enabled, you'd be warned about the fact when you tried to create an executable version of your application. The reason for the warning is quite simple: "Enable runtime checks of external references?" does the same job as "Fully qualify external references?" except that the latter is enforced at compile time, while the former is applied at runtime. It sounds obvious, but it's worth pointing out that although compile-time checks have no effect on runtime performance, runtime checks do. If you've completed testing of your application, you should disable runtime checks in the production version.

I recommend that you set "Fully qualified external references?" to "Yes" and "Enable runtime checks..." to "No." In addition, of course, qualify every reference necessary to uniquely identify a variable.

Which beings me to global variables. With the advent of window parameters and static variables, the need for global variables is virtually removed. A suggested rule: Define only those global variables that must exist and can't be defined at any other level of the application (that won't be many).

If global variables don't exist, they can't take precedence over unqualified references; thus, this hidden "feature" of SQLWindows won't often bother you. By following these guidelines, you can safely assume that any unqualified reference is a local reference; the variable or object being referenced is declared on the object or on the object's container.

It sounds so simple . . .

The examples I've used so far concentrate on relatively simple variable references, those referring to variables declared on a form or dialog. What about classes? What about function variables? What about object references?

When a reference is made to any variable or object, the compiler first checks the local possibilities and works outward and upward from there. In the case of a function making a variable reference, first, function variables and parameters are checked, then the variables and parameters of the window or class the function is a part of, and finally the global variables. If no variable can be found, then it's deemed to be an external reference and everything I stated earlier is still true. (But, of course, we all now fully qualify our external references, don't we?)

Let's change the rules

There's one further complication, however. If you dig around in the reference manuals or the on-line help long enough, you'll come across the following functions:

- SqlVarSetup( )
- SqlImmediateContext( )
- SqlContextSet( )
- SqlContextSetToForm( )
- SqlContextClear( )

Notice that the latter three of these functions have been superseded by the first function listed; consequently I won't discuss them—suffice it to say they have limitations that the more recent enhancement, SqlVarSetup( ), was created to solve. You may run across the various SqlContext* functions in your maintenance work, but you shouldn't use them in new development. Similarly, I won't mention SqlImmediateContext( ), though for a slightly different reason; I never use the related SqlImmediate( ) function, and never intend to. I always use SqlPrepareAndExecute( ) and a single SqlFetchNext( ) when necessary to ensure that I know which Sql Handle is being used and, consequently, to be sure of which database I'm accessing in what isolation level with what message buffer sizes and so on.

If you consult the on-line help about SqlVar-Setup( ), you'll probably come away confused. Don't be. The essence of SqlVarSetup( ) is relatively straightforward and applies most often to implementations of classes and global functions.

Normally, when you SqlPrepare( ), SqlExecute( ) or SqlFetchNext( ), SQLWindows attempts to resolve any variable references in your SQL statement using the same old set of tried and tested rules that I laid out earlier. For bind variables this happens during the execution of your SQL. For select INTO variables this is done at fetch time. Neither is always ideal.

The basic function of SqlVarSetup( ) is to fool the SQLWindows runtime into thinking that it's executing and/or fetching the SQL associated with a given handle in a different "execution context." This is probably best explained by an example—notably lacking from the on-line help.

Declare two global variables as follows:

```
◇ Sql Handle: hSql
◇ Number: nFetch
```

And implement two "auto-create" form windows:

```
◆ Form Window: frmPrepare
  ◆ Contents
    ◆ Pushbutton: pbGetName
      ◆ Message Actions
        ◆ On SAM_Click
          ◇ Call Prepare( )
          ◇ Call SqlVarSetup( hSql )
          ◇ Call frmFetch.ExeAndFetch( )
    ◇ Data Field: dfTable
  ◆ Functions
    ◆ Function: Prepare
      ◆ Actions
        ◇ Call SqlPrepare( hSql,
          'SELECT NAME FROM SYSTABLES INTO :dfTable' )
  ◆ Message Actions
    ◆ On SAM_Create
      ◇ Set SqlDatabase = 'DEMO'
      ◇ Call SqlConnect( hSql )
    ◆ On SAM_Destroy
      ◇ Call SqlDisconnect( hSql )

◆ Form Window: frmFetch
  ◆ Contents
    ◇ Data Field: dfTable
  ◆ Functions
    ◆ Function: ExeAndFetch
      ◆ Actions
        ◇ Call SqlExecute( hSql )
        ◇ Call SqlFetchNext( hSql, nFetch )
```

Observe the behavior of the application with and without the call to SqlVarSetup( ). With SqlVarSetup( ) called as in the preceding example, the first table name is fetched into frmPrepare.dfTable, *not* frmFetch.dfTable! Even though the runtime doesn't resolve the reference to dfTable until the SqlFetchNext( ) function is called, SqlVarSetup( ) has "pinned" the SQL execution to frmPrepare.

There's no external reference to resolve with respect to dfTable since a local symbol exists on each form.

It might not be obvious, but each call to SqlVarSetup( ) relates *only* to the most recent SqlPrepare( ) for the associated SQL handle. This can be shown by changing the code on the push button in the previous example to the following: (Again, it's a contrived example, but I hope it serves to illustrate the point.)

```
◆ Pushbutton: pbGetName
  ◆ Message Actions
    ◆ On SAM_Click
      ◇ Call Prepare( )
      ◇ Call SqlVarSetup( hSql )
      ◇ Call Prepare( )
      ◇ Call frmFetch.ExeAndFetch( )
```

Remember that BIND variables get evaluated at execution time and INTO variables at fetch time; so if SqlVarSetup( ) is called after the *execute* rather than the prepare, BIND variable references will be resolved in a potentially different context than the INTO variables!

Summary

The fundamental rules of scoping in SQLWindows are less confusing than they might at first appear. Far from causing problems, the rules can be made to work *for* you, even if it does take a little help from the compiler and a bit of discipline from you, the developer. **P**

Jolyon Smith has worked with SQLWindows for four years in a variety of environments, from software and systems houses to financial and manufacturing companies. Jolyon is currently a SQLWindows consultant and trainer for PLATINUM technology. CompuServe 72660,617, Internet smithjt@dircon.co.uk.

Get Connected with Hierarchies

Michael Cunningham

Many sets of data in the real world are organized into hierarchies: company organization charts, bills of material, family trees, and others. The Visual Toolchest offers a way to portray such data graphically in SQLWindows application. But what's the best way to store such data in databases and load it into tree-structured list boxes? The author presents an effective answer for Oracle databases, with general interest to all SQLWindows developers.

In this article, we're going to create a list box class that, after initialization, is capable of populating and displaying the data in a hierarchical tree. I'll need to derive a new class from cOutlineListBox and add the functionality specific to handling hierarchical data. I'll call the new class cOutlineConnectBy (hereafter referred to as OCB) and the examples throughout the article will refer to data in an organization chart (org chart). I'll display the org chart data graphically in the list box. (See Figure 1.) The source code for this class, available through the services listed on Page 2, can be dropped into your applications for immediate productivity improvements.

The SQLWindows Visual Toolchest's list box class cOutlineListBox, and an Oracle SQL extension commonly referred to as the "CONNECT BY," are particularly well suited for working with hierarchical trees such as a family tree or an organizational chart. The cOutlineListBox is a list box class that provides functionality similar to that of the list boxes in Windows' File Manager. The CONNECT BY is a clause added to a SELECT statement to make querying hierarchical data simple. Like most other databases, standard ANSI SQL doesn't support CONNECT BY for querying hierarchical data.

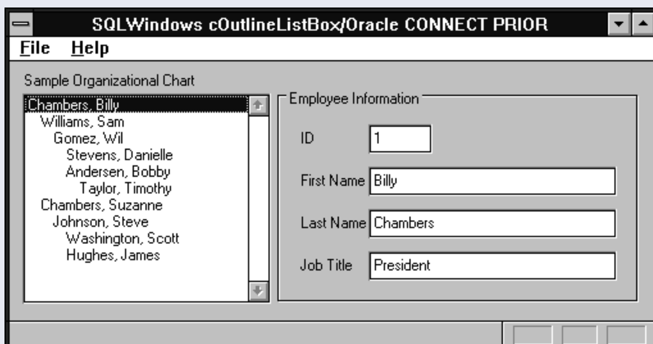

Figure 1. The creation of this organization chart demonstrates how to use cOutlineConnectBy in working with hierarchical data.

What's in a tree?

A hierarchical tree is a directed graph made up of nodes. In my examples I'll discuss an org chart in which each node in the tree represents an employee. The top node in the tree is called the root node, and the nodes connected to the root are called child nodes (making the root a parent node). The nodes without children are leaf nodes (an employee with no subordinates). In the remainder of this article, we'll use the family tree analogy of grandparent, parent, sibling, and child.

The outline list box

You should know a few things about outline list boxes before managing the items in the list. The outline list box class displays a hierarchical tree and therefore needs a way to manage each node in the tree (or item in the list box). Each item in the outline list box has a set of properties you use to manage your hierarchy. The ones to be concerned with are: *Item Handle, Item Text, Parent Handle, and Item Value*.

The Item Handle is a unique identifier given to each item (node) in the list box. The Item Handle is to items in the outline list box as indexes are to items in a standard list box. The difference is this: Because the Item Handle is actually a direct pointer to a node in the tree, it always remains the same. For example, the outline list box supports Drag and Drop to enable a user to move items around in the tree's hierarchy. If a user were to drag a list box item and move it up in the list, it would have a different index than it did previously, but the Item Handle would remain the same.

Since all nodes in hierarchical trees are connected (the connections are called edges), each node needs to know which existing node to attach itself to when added to the tree. In other words, it needs to know the Item Handle of its parent. The Visual Toolchest refers to this as the Parent Item Handle and uses it as a parameter to the function that adds the item to the list. This is how all the items get

connected, thereby forming the tree.

Figures 2 and 3 show a grandparent box as the root node. The first parent box is a child node to grandparent. The second parent box can be thought of in two ways: it is a child of grandparent, but also a sibling to the first parent. Items can be added to the list box as either siblings or children. In the case of the second parent box, it could have been added as either a child to Grandparent or as a sibling to the first parent. The results would be identical; the parent of the second parent would be the grandparent. When you add the item as a sibling, Visual Toolchest takes care of assigning the correct parent node for you.

One last thing to note is that each item in the outline list box can have a 32-bit numeric value associated with it. This is actually a Windows feature for all items in list boxes, and Visual Toolchest provides an easy method for setting those values. Since you'll populate your list box with an org chart, and each item in your list box will be an employee, the value associated with each item will be the employee's I.D. number. This makes it easy to query for additional data about each employee when an item is clicked on. If you're doing new database design for hierarchical data, it's advisable to assign a column to hold a unique, numeric key for each row. That column will allow you to take advantage of the list box feature.

Our sample tree

As noted above, our sample application, VT_ORA.APP, will use an org chart. File VT_ORA.WTS contains the script to build table t_employee, which looks like Table 1.

Oracle's CONNECT BY

An org chart is hierarchical data that can be represented by a tree. Oracle has provided a mechanism for querying hierarchical data, commonly referred to as "CONNECT BY". Two clauses aid in constructing queries, CONNECT BY and START WITH.

The CONNECT BY clause specifies the relationship between parent rows and child row. While executing the query, CONNECT BY tells Oracle to find the next record

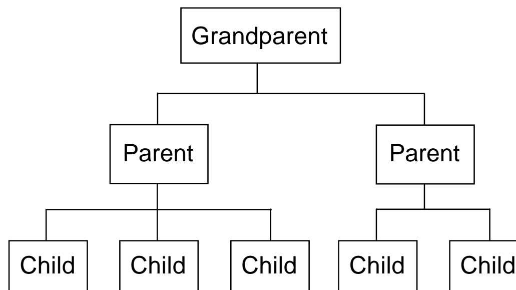

Figure 2. A simple example of a hierarchical tree. Items can be added as either siblings or children. In the case of the second parent box, it could have been added as a child to the grandparent box (or root node) or as a sibling to the first parent box.

where the PRIOR column is equal to a given value. In our application the column referenced by the PRIOR keyword is the manager_id and the given value is the current row's employee_id. In effect, Oracle will look for a row where the manager_id is equal to the current row's employee_id. For instance, if the current row were the president of the company, Oracle would look for a row where the manager_id was the same as the president's employee_I.D. There might be several such rows in the database, but Oracle would fetch only the first one. Then it would search for a row which was the "child" of the row it just fetched (a "grandchild" of the president). If found, it would then look for children of *that* row, and so forth. In other words, CONNECT BY causes the fetch to be *recursive*—to go completely down each branch and sub-branch of the tree before returning to fetch data from other major branches. Because of this, the physical sequence of rows in the result set returned to SQLWindows is the exact sequence you need to load the list box.

Table 1. The table t_employee.

employee_id	manager_id	first_name	last_name	job_title
1	0	Billy	Chambers	President
2	1	Sam	Williams	Vice President Manufacturing
3	1	Suzanne	Chambers	Vice President Marketing
4	2	Wil	Gomez	Manufacturing Director
5	8	Timothy	Taylor	Manufacturing Worker
6	3	Steve	Johnson	Marketing Manager
7	4	Danielle	Stevens	Manufacturing Supervisor
8	4	Bobby	Andersen	Manufacturing Supervisor
9	6	Scott	Washington	Marketing Staff
10	6	James	Hughes	Marketing Staff

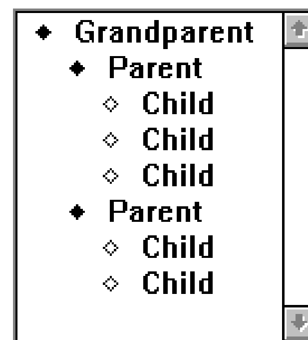

Figure 3. Here, the hierarchy has been converted into an outline list box.

The START WITH clause specifies the root row or rows of the hierarchy. This would be like asking Oracle to return all the employees under a supervisor. As long as you knew the supervisor's employee I.D., you could build a hierarchical result set of every employee "below" the supervisor. Since the CONNECT BY is recursive, it takes care of returning the rows in the result in the exact order you'd use to add them to the list box. By starting the query with a supervisor's I.D., you'd return a subset of the entire organization. In our application, we have omitted the START WITH clause to get the entire set of hierarchical data starting from the root node.

A sample SQL statement using CONNECT BY looks like this:

```
SELECT LEVEL, last_name, first_name
FROM employees
CONNECT BY PRIOR employee_id = manager_id
START WITH employee_id = 1;
```

LEVEL in the statement above is an Oracle *pseudocolumn*, like ROWID and SYSDATE. When processing the query, the LEVEL pseudocolumn returns 1 for the root node of the hierarchical tree, 2 for the children of the root node, 3 for the grandchildren, and so on. LEVEL plays an important role when it comes to populating your OCB list box, because you'll need to know if a new list box item needs to be added to the previously added item as a sibling or child. If the LEVEL of one row were the same as the previous row, then you'd know to add the new item as a sibling of the previous item. Likewise if the LEVEL were higher, you'd want to add the new item as a child to the previous item.

Before continuing, I'd like to explain that this list box will be populated with the entire org chart hierarchy *at one time*. I've seen many ways of populating an outline list box and one of the most popular is to start by populating the list box with only one level of data—like the president of the company. As the user double-clicks on the president, the database is queried for all the employees who reported to the president and the new rows are added under the president as children. As one of those items is double-clicked, the same would occur again. This is fine for most databases that don't support hierarchical queries, but since Oracle provides a more efficient way of querying the data, I suggest you use it. And by doing so, you can populate the entire list box at one time.

Functional goal for cOutlineConnectBy

I want a list box class that can be set up with one function call (the initialization). Then I want to be able to tell the list box to populate itself. The population actually uses two steps. The first step involves populating an array of structures (array Item in class cOutlineConnectBy in the sample application VT_ORA2.APP) with the result of my query. The second step involves using the data in the array Item to populate the OCB.

After population you'll want the OCB to provide some information each time the user clicks on an item in the list box. The information you'll want is the value associated with the item chosen, employee I.D. number in this case. Also, in case there's more than one OCB list box on a form, have it give its window handle. For both pieces of information you'll use a message. When the user clicks on an item, you'll send a PM_OutlineClick click message to the OCB list box's parent window. wParam is the window handle of the list box and lParam is the value associated with the item clicked on. At the form level you can do what you want with the information from the message.

This process creates a somewhat generic class tailored for population from an Oracle database. The truth is that as long as array *Items* is populated correctly, the population and functionality of the OCB remain the same regardless of whether array Items were populated from Oracle or SQLBase or any other SQL database.

What About Windows 95?

If you attended the Gupta Developers Conference last June, you may remember a presentation titled "Getting Ready for Windows 95." It served as a reminder that any coding technique you consider adopting today must be viable tomorrow under Windows 95 and the 32-bit versions of SQLWindows.

So, how well do the techniques presented in this article score on that basis? Quite well. Windows 95 has an application called Explorer, which makes extensive use of tree-structured list boxes. These boxes have some additional display capabilities beyond those available in Windows 3.1. At the Developers Conference it was announced that the next version of SQLWindows' Visual Toolchest will feature a list box that emulates those in Explorer.

The good news for developers is that the Explorer object is basically the same as the current cOutlineListBox, with some behavioral extensions. The extensive use of small bitmaps in Explorer list boxes is already supported by SetItemPicture and related functions within cOutlineListBox. Thus, coding techniques which work well with cOutlineListBox will work well in future versions of SQLWindows.—*Mark Hunter*

Creating the cOutlineConnectBy class

Initialization is going to be the trickiest part of using the OCB list box class, because you need to

provide all the pieces necessary for the class to be able to build the CONNECT BY SQL statement. Although the internals of our initialization function are complicated, it is a tested, adaptable function which will work well for you with just an understanding of the parameters it uses. For the initialization of the OCB list box, you'll have to provide a function called InitParms( ):

```
[hd]Call InitParms( 't_employees',
        'last_name', 'employee_id',
        'manager_id', 'employee_id',
        'employee_id', '1' )
```

The InitParms( ) function receives several parameters:

- The name of the table to use in the query.
- The column name in the table that holds the item text you want displayed in the list box.
- The column name in the table that holds the numeric value associated with the list box item.
- The first column in the CONNECT BY clause. Oracle uses this column as the key for searching for child rows of the current row.
- The second column in the CONNECT BY clause. Used to connect a child row to a parent row.
- The column in the table to use as the START WITH column.
- The value to start with on the START WITH column.

The resulting SQL statement is shown below. It's concatenated in a function in which the INTO variables are defined.

```
SELECT level, last_name, employee_id
INTO :fnLevel, :fsItemText, :fnItemValue
FROM t_employees
CONNECT BY PRIOR employee_id = manager_id
START WITH employee_id = 1
```

For populating the OCB list box there are two functions, Load( ) and Populate( ). Load( ) receives a SQL handle to be used for querying the database. I made the choice to provide a SQL handle so that I could set the SQL handle parameters and isolation level if I needed to Load( ), where the concatenation of the SQL statement pieces takes place, uses the instance variables set by the InitParms( ) function to write the SQL.

Load( )'s most important task is to load the array *Items* with the data needed to populate the OCB list box. *Items* is an array of user-defined variables from class cListBoxItems. The instance variables in the functional class are:

```
Number: Level
    ! The level the current item will be when
    ! added to the list box. This variable comes
    ! from the 'LEVEL' in the SQL statement.
String: Text
    ! The item text to show in the list box when
    ! populated.
Number: Value
    ! The value associated with the item in the
    ! list box.
```

Whole Lotta Shakin'…

Continued from page 2

resources for a major pitch for Centura. I'd like to see the company say that Centura is the best toolset on the market, period–and say it loudly and often.

What about poor old 16-bit SQLWindows? Gupta has said in so many words that there won't be any significant new development for the 16-bit version. However, they have solicited "wish lists" from beta testers asking what features of the beta would be most appreciated if they were ported backward to a maintenance release following 5.0.2. So we can anticipate at least one maintenance release, featuring some undisclosed set of goodies from the Centura development effort. Beyond that, there's no indication of another version.

Will Centura be a revolutionary improvement over SQLWindows? Will Gupta's marketing for Centura be a revolutionary improvement over their current marketing? Will Centura attract new customers to the Gupta community rather than simply extracting additional money from existing customers? We at Pro Publishing don't know, but we're committed to finding out the answers as quickly and thoroughly as possible. *Gupta Pro* is a technical newsletter first and foremost, but we will also track the major events affecting the careers of all developers using Gupta products.

News of Umang Gupta's retirement prompts these thoughts: He inspired and directed the creation of the best client/server toolset in the world. Without his hard work and foresight, we developers would not be arguing now about whether the company should be acquired or how SQLWindows should be marketed; instead we'd be cursing Visual Basic or struggling with GPFs in PowerBuilder. Even if changing times and markets now demand different leadership, it doesn't lessen the significance of his work. I thank him and wish him well.

I hope you enjoy the premier issue of *Gupta Pro*. In these turbulent times there's a great need for information and support among Gupta's customers. The many talented, generous developers who contribute to the GUPTA forum on CompuServe have done much to meet that need. Now Pro Publishing is here to help, too. Contact us, let us know what you're looking for, and we'll work hard to provide it. **P**

While populating the OCB list box with the functional class array, each variable plays a part. As I said earlier, as long as the functional class array is populated correctly, the population and functionality remains the same.

For the OCB list box class to work correctly with the Oracle database, the table you're querying needs to conform, in part, to a standard schema. The schema columns needed to support the OCB list box class are defined as:

```
CREATE TABLE org_chart(
 employee_id numeric NOT NULL,
 manager_id numeric NOT NULL,
 last_name varchar2( 30 ) );
```

Of course, the names of the columns don't have to conform to this schema, and there can be other columns defined, but this is the general structure your table needs to follow. The employee_id and manager_id are needed for the CONNECT BY; the last_name gets used for the list box item text. The LEVEL is obtained from the pseudocolumn provided by Oracle. The employee_id, in addition to being used in the CONNECT BY, also is the value to be associated to the OCB list box item.

The second phase of populating the OCB list box is to use the functional class array and add each item to the OCB list box. The function provided for the population is simply called Populate( ).

When adding items to the list box, give an Item Handle of an existing item that will act as a parent or sibling for the new item. For this you need to do something special. To start, the Visual Toolchest cOutlineListBox class provides a root node. This special list box entry is never visible and can't have its value changed. The Item Handle of the Root node is obtained with the GetRoot( ) function in the class. You can use it in the OCB list box because of inheritance.

I mention the cOutlineListBox root node because your hierarchical tree created with the CONNECT BY gives you one or more visible "root" nodes. For example, if a company had two top officers, both of them would have NULL values in their "manager_id" columns. In this situation, they would both appear as top-level items in the list box, with various employees indented underneath them. The sample data sticks with the more common situation of a single top officer. So the visible "root" node in the tree is actually a child node of the root node provided by the Visual Toolchest's cOutlineListBox class.

Keep track with a stack

Populate( ) has an important task while populating the OCB list box. Recall that I said earlier you need to know the LEVEL of the item you're adding to the OCB list box to know if you should add it as a child or a sibling. For this I chose to use a stack of items.

The stack is a parallel array representing the most recent items added to OCB list box for a particular level. The stack keeps track of items by their level and their Item Handle. Only one item at a time of a particular level will ever be found on the stack.

If the item at the top of the stack has the same LEVEL value as the new item you're adding, add the new item as a sibling. After adding the new item, pop the top item off the stack and push the new item on the stack. This keeps the most recent item for this level on the stack. The previous item for this level is no longer needed on the stack because no other items will need to be added to it.

Likewise, if the level of the new item you're adding is greater than the top item on the stack, simply add the new item as a child to the top stack item and push the new item on the stack.

Summary

The outline format is a popular and effective way to present information, and will become even more important with the introduction of Windows 95 and its Explorer-style windows. Visual Toolchest's cOutlineListBox class makes your visual presentation task easier, and Oracle's CONNECT BY feature greatly simplifies your data retrieval task. The cOutlineConnectBy class introduced here and in the online disk files can serve as a solution for many of your tree-structured hierarchy presentations. **P**

Download CUNNING1.ZIP (GO GUPTA, Library 10).

Michael Cunningham is an independent consultant in Southern California specializing in SQLWindows development. He's also a member of Gupta's TeamAssist, a volunteer group of developers providing help on Gupta's CompuServe forum. 310-860-9210, CompuServe 75112,2617.

Static Control . . .

Continued from page 6

```
◇ Set hStatic = SalStaticGetLabel( df1 )
```

Final notes

With this article and the accompanying sample applications you should have a good understanding of how statics work and behave in the SQLWindows environment. You also have the tools to add the fine touches to your applications that may improve its aesthetics without sacrificing performance. **P**

Download BURKE1.ZIP (GO GUPTA, Library 10).

R. J. David Burke is a Senior Consultant with Gupta Professional Services. He's coauthor of *Special Edition, Using Gupta SQLWindows 5* (Que, 1995) and a Gupta Certified SQLWindows instructor.